UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Double degree in Computer Science and Mathematics

DEGREE WORK

scikit-fda: Principal Component Analysis for Functional Data

Author: Yujian Hong
Advisor: Alberto Suárez González

May 2020

Yujian Hong
scikit-fda: Principal Component Analysis for Functional Data

Yujian Hong

PRINTED IN SPAIN

Agradecimientos

En primer lugar, quería agradecer a Alberto Suárez, mi tutor, por todo el apoyo durante el año al trabajo y por las ideas que me dan en las reuniones para mejorar el trabajo. También quiero agradecer a Carlos Ramos por revisar mi código y apoyarme en los asuntos técnicos de la librería scikit-fda.

En segundo lugar, quería agradecer a mi familia dado la situación extraordinaria causada por el COVID-19. Gracias por el silencio en mis horas de trabajo y la inmensa paciencia que han tenido conmigo.

Resumen

El análisis de datos funcionales (FDA por sus siglas en inglés) es una rama de la estadística que estudia datos obtenidos de funciones que dependen de parámetros continuos, como por ejemplo la variación de la temperatura a lo largo del año o el crecimiento humano. Los datos obtenidos son una discretización de las funciones en ciertos puntos del dominio de las funciones, lo que comúnmente se denomina la representación discreta de los datos. Otra forma de expresar los datos es como combinaciones lineales de otras funciones que forman una base en el espacio original, denominado representación en bases.

En este trabajo, se estudia una técnica de reducción de dimensionalidad frecuentemente usada en la estadística multivariante llamada análisis de componentes principales (PCA por sus siglas en inglés). Consiste en proyectar los datos originales en nuevas direcciones que maximizan la varianza poblacional, denominadas componentes principales. La reducción de dimensión se produce porque solo se necesita un número reducido de componentes para conservar la mayoría de la varianza.

El objetivo principal de este proyecto es entender cómo se puede aplicar este método en el contexto funcional. Nos centramos en las particularidades que se producen al tratarse de datos funcionales. También se estudian las diferencias entre la aplicación de este método a las dos representaciones de los datos funcionales.

Otro aspecto a considerar es que los datos originales pueden contener ruido, que a su vez causa irregularidades en los componentes principales. El proceso denominado regularización, que consiste en suavizar los componentes eliminando el ruido no deseado, también se estudia en este trabajo.

La mayoría de las librerías que implementan funcionalidades de FDA se han realizado en el lenguaje R. Dado el aumento de la popularidad de Python en los últimos años, en 2018 se inició el desarrollo de un paquete de código abierto llamado *scikit-fda* [8] para facilitar el análisis de datos funcionales en este lenguaje. En este trabajo, el análisis de componentes principales para ambas representaciones de los datos funcionales ha sido añadida a esta librería. Usando esta implementación, se han analizado varias muestras de datos reales para mostrar las aplicaciones prácticas de este método.

Palabras clave

Análisis de datos funcionales, reducción de dimensionalidad, análisis de componentes principales, código abierto, Python

Abstract

Functional data analysis (FDA) is a branch of statistics that studies data obtained from underlying functions that depend on continuous parameters, such as temperature variations or the human growth curve. The resulting data is a discretization of the functions at certain points of the continuous parameter, and this is called the discrete representation of the data. We can express the original data also as linear combinations of other functions that form a basis in the original space. This is called a basis representation.

In this work, we study a dimensionality reduction technique widely used in multivariate statistics called principal component analysis (PCA). PCA consists in projecting the original data into new axes that maximize the population variance, named principal components. The dimensionality is reduced because only a few components are needed to capture most of the original variance.

The main objective of this project is to understand the application of this method in the functional context, focusing on the particularities that functional data present with respect to the multivariate case. Within the functional context, differences also arise when we deal with the two representations of functional data.

Another aspect to consider is that the original data may contain noise which causes irregularities in the principal components. It is possible to smooth the components when obtaining them, eliminating the undesired noise. This process is called regularization.

Most libraries that deal with FDA are implemented in the R language. Given the increasing popularity of Python, an open-source package called *scikit-fda* [8] was created in 2018 to support FDA. In this work, functional principal component analysis for both representations is added as an extra feature to this open-source package. Using this implementation, we analyzed several real-world datasets to show the practical applications of this method.

Keywords

Functional data analysis, dimensionality reduction, principal component analysis, open-source, python

Table of Contents

1 Introduction 1

1.1 Goals and scope . . . 2

1.2 Document structure . . . 2

2 State of the art: functional principal component analysis 3

2.1 Multivariate PCA . . . 5

2.2 Definition of FPCA . . . 5

2.3 FPCA, eigenvalues and eigenvectors . . . 6

2.4 The importance of FPCA . . . 8

2.5 FPCA for discrete representation of the data . . . 9

2.6 FPCA for basis representation . . . 10

2.7 Regularized FPCA . . . 12

2.8 Depth based median and trimmed mean . . . 17

3 Analysis, design and development 19

3.1 Analysis . . . 19

3.2 Design . . . 20

3.3 Coding . . . 23

3.4 Documentation and unit tests . . . 23

3.5 Development, version control and continuous integration . . . 24

4 Results 27

4.1 Berkeley Growth Study . . . 27

4.2 AEMET weather data . . . 30

4.3 Phonemes dataset . . . 32

5 Conclusions and future work 37

Appendices 41

A Trimmed mean and depth based median for the dataset Phonemes 43

A.1 Overall trimmed mean for the dataset Phonemes . . . 43

A.2 The trimmed mean for each phoneme . . . 44

B Simulation result notebook 47

C FPCA Documentation 87

Lists

List of equations

2.1 Discrete representation of a functional data sample . . . 3
2.2 Basis representation of a functional data sample . . . 4
2.3 Gram matrix . . . 4
2.4 Multivariate principal component scores . . . 5
2.5 Multivariate maximization condition in PCA . . . 5
2.6 Multivariate orthogonal restriction for subsequent components . . . 5
2.7 Functional principal component scores . . . 6
2.8 Functional maximization criterion for PCA . . . 6
2.10 Orthogonal restrictions for the subsequent components of FPCA . . . 6
2.11 Multivariate maximization problem for PCA, written in matrix notation . . . 7
2.12 Eigenequation for multivariate PCA . . . 7
2.13 Covarirance Operator . . . 7
2.14 Covariance function . . . 7
2.15 Functional PCA eigenequation expanded . . . 7
2.16 Functional PCA eigenequation . . . 8
2.17 Functional PCA final eigenequation . . . 8
2.18 Eigenequation of FPCA for the discrete representation . . . 9
2.19 Principal component scores for discrete representation . . . 10
2.20 Eigenequation for basis representation . . . 11
2.21 Principal component scores in basis representation . . . 12
2.22 Penalty of a arbitraty curve . . . 13
2.23 Penalized sample variance . . . 13
2.24 Orthogonality restriction in regularized FPCA . . . 14
2.26 Penalty of a arbitrary curve for discrete representation . . . 14
2.27 Generalized penalty of a arbitrary curve in discrete representation . . . 14
2.28 Regularized eigenequation for discrete representation . . . 15
2.29 Summary of FPCA . . . 16
2.30 LOO-CV error for an individual data and a fixed number of components . . . 16
2.31 LOO-CV error for a fixed number of components . . . 16
2.32 LOO-CV error . . . 16

List of figures

3.1 Overall structure of scikit-fda. 20
3.2 Gitflow model. 25

4.1 Ten growth curves from the Berkeley Growth Study. 27
4.2 The first two principal component curves of the Berkeley Growth Study, in both representations 28
4.3 The perturbations over mean for the first two principal components of the Berkeley Growth Study 28
4.4 Dispersion diagram of the first two principal components of the Berkeley Growth Study, with logistic regression decision boundary. 29
4.5 Five curves from the AEMET temperature data, from 5 chosen stations. 30
4.6 First two principal components of the AEMET weather dataset 31
4.7 Perturbations of the mean for the first two principal components for the AEMET temperature data. 31
4.8 Dispersion diagram for the AEMET temperature data, dividided into 5 clusters. 32
4.9 Stations mapped to the Spain map and divided in 5 clusters. 33
4.10 Ten curves of the Phonemes dataset, two of each phoneme. 34
4.11 First principal component of the Phonemes dataset, in both basis and discrete representations. 35
4.12 First principal component of the Phonemes dataset, regularized for different regularization parameters. 35
4.13 Differences between the regularized first principal component, and the original principal component, in both representations. 36
4.14 Comparison of regularized curve in three different ways. 36

A.1 The overall mean and individual mean of each phoneme. 43
A.2 Trimmed means for the Phonemes dataset, with different trim percentages. 44
A.3 The trimmed mean functions for each phoneme compared with original mean functions. 45
A.4 50 random dcl data curves before and after trimming 45
A.5 Comparison of median curves with mean curves for the Phonemes dataset. 46

1

Introduction

Functional data analysis (FDA) is a branch in Statistics that studies data that are functions. Namely, objects that depend on a continuous parameter and can take infinitely many values. The continuous parameter is often time, but it can be any other relevant variable such as frequency, light intensity, etc. FDA is a relatively young field that started the mid-twentieth century, since then the interest in this branch has risen considerably because of its application in fields such as medicine, physics, environmental sciences, etc. One common reference is *Functional Data Analysis* [17], where the main ideas of FDA are discussed.

The majority of the packages that support this field can be found in R or Matlab, such as `fda` [18] or `fda.usc` [6]. Due to the increasing usage of Python in the context of Statistics and Machine Learning, an open-source project in Python called `scikit-fda` [8] was started in 2018 by Miguel Carbajo [1] as part of his degree work. The aim is to provide support for FDA in Python and make this powerful tool available to the growing Python scientific community. Since then, the library is being developed by the Machine Learning Group at the Universidad Autónoma de Madrid together with undergrad students from said university with interest in the area. The aim of this project is to continue the development of the library `scikit-fda`.

When we obtain data, one key aspect to consider is the number of attributes extracted. An increasing number of attributes entails an exponential growth of the data space, and the available data soon becomes sparse and appears to be different in many ways. This effect is known as the curse of dimensionality, and it causes several problems in data classification as well as data analysis. This is especially important in FDA because the data considered are functions that live in an infinite-dimensional space.

Dimensionality reduction methods are considered to address this problem. One of the most commonly used in multivariate statistics is principal component analysis (PCA) [20]. The idea is to express the data in new variables that are orthonormal linear combinations of the original variables. The new variables are called principal components, which are obtained maximizing the sample variance and in descending order by the amount of variance explained. This step is often accomplished by using a projection matrix, as only linear transformations are considered. The reduction of dimensionality occurs because only a few principal components are needed to represent the original data while preserving

most of the sample variance. An important assumption made in PCA is that the information lies in the sample variance, therefore, the method is only appropriately used when this is the case. The same idea can be applied for FDA, and the resulting method is called functional principal component analysis (FPCA).

1.1. Goals and scope

The main goal of this project is to extend the functionalities already implemented in `scikit-fda`, incorporating FPCA in the library. The similarities between multivariate and FPCA are examined as part of this goal as well as the differences of this method when applied to functional data in different representations.

However, this is not the only feature implemented, other features related to FDA are also incorporated. This includes trimmed mean and depth based median for functional data, which are exploratory methods for functional data.

The software is open source and intended for a general audience. To achieve a high code quality, rigorous code standards and reviewing are applied. Also, thorough testing is done to ensure code robustness. As `scikit-fda` is a package designed to align with `scikit-learn` [15], compatibility with the standards established by said library is followed in the coding process. In order to facilitate the use of the library as it is intended to reach a wide audience, the implemented software is extensively documented.

1.2. Document structure

The main part of this document is divided into four chapters, excluding the introduction. In Chapter 2 the state of the art about FPCA is shown, which gives an overview of the mathematical basis, along with some preliminary concepts in FDA to aid the understanding of FPCA.

In Chapter 3 the analysis, design, and development process is given, explaining the technical decisions in detail. In Chapter 4 several examples are analyzed with FPCA, in order to illustrate the practical applications of this method. Finally, Chapter 5 presents the main conclusions of this project and some remarks about the future development of FPCA related work in the package.

In addition to the main document, there are 3 appendices. In Appendix A an analysis of one dataset used in Chapter 4 is performed using functional trimmed mean and depth-based median. In Appendix B the notebook containing the code used to obtain the figures used in Chapter 4. In the last appendix the documentation of the implemented FPCA module is included.

2

State of the art: functional principal component analysis

In FDA, the analysed data comes from measuring functions, curves or surfaces that vary over a continuum. The hypothesis is they live in the functional space L^2 of square integrable functions. For the rest of this work, these functions varies over a closed interval $\mathcal{T}$. The L^2 space has an associated operation called the inner product that is defined as $\langle x, y \rangle_2 = \int_{\mathcal{T}} x(t)y(t)dt$. We also define the norm as $\|x\|_2 = \langle x, x \rangle_2^{\frac{1}{2}}$.

In theory, we should be able to record infinite values as a function that can be evaluated at any given point in the interval. In practice, however, it is not possible due to computation and storage limits. Then the problem of how a functional data can be represented arises from the fact that we cannot recover directly the underlying function. The functional data is then measured only at certain points, and we use $\mathbf{t} = (t_1, t_2, ...t_M)^T$ to denote the vector with said points, where M is the number of points.

Let N be the number of functional data in a data sample, and $\{x_i(t)\}_{i=1}^N$ be the underlying functions from which we obtain the data. From each function $x_i(t)$ a vector $\mathbf{x}_i$ of length M is obtained. Then we can denote the matrix representing the data using $\mathbf{X}$:

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & ... & x_{1M} \\ x_{21} & x_{22} & ... & x_{2M} \\ \vdots & & & \\ x_{N1} & x_{N2} & ... & x_{NM} \end{pmatrix}, \tag{2.1}$$

This is one possible representation of a functional data sample, and throughout this work, this representation will be referred to as the discrete representation. One may wonder what makes FDA different from multivariate statistics as the obtained data is represented in the same way. The main difference is that the high correlation between the values x_{ij} and $x_{i(j+1)}$ due to the functional nature of the data is not taken in to account in multivariate analysis. While in FDA the main focus is on this functional nature, allowing the usage of functional analysis techniques such as integration or derivation. Another difference is that in FDA it is possible to evaluate a functional datum $x_i(t)$ at any point t in the defined interval by interpolation.

Another possible representation of a functional data sample is by expressing them as linear combinations of basis functions, and it is called the basis representation. The coefficients of the linear combinations are obtained by projecting the underlying functions into the subspace formed by the basis functions. We use $\{\phi_i\}_{i=1}^{i=K}$ to denote the basis with k functions, that can be the truncated Fourier basis, the BSpline basis, or the polynomial basis, etc. The BSpline basis contains polynomials defined at certain intervals. The election of the basis is important in order to make the resulting projection similar to the original functions. For example, for periodic functions, it is better to use the truncated Fourier basis.

After writing the original functions in the chosen basis, we might write the functional data sample as the following matrix:

$$\boldsymbol{\xi} = \begin{pmatrix} \boldsymbol{\xi}_1^T \\ \boldsymbol{\xi}_2^T \\ \vdots \\ \boldsymbol{\xi}_N^T \end{pmatrix} = \begin{pmatrix} \xi_{11} & \xi_{12} & \dots & \xi_{1K} \\ \xi_{21} & \xi_{22} & \dots & \xi_{2K} \\ \vdots & & & \\ \xi_{N1} & \xi_{N2} & \dots & \xi_{NK} \end{pmatrix}. \tag{2.2}$$

However, we have to keep in mind that $\boldsymbol{\xi}_i^T = \begin{pmatrix} \xi_{i1} & \xi_{i2} & \dots & \xi_{iK} \end{pmatrix}$ is not the function we want to analyse, it is only the coefficient vector in the chosen basis. The approximated function is $\xi_i(t) = \sum_{j=1}^{K} \xi_{ij}\phi_j(t) = \boldsymbol{\xi}_i^T\boldsymbol{\phi}$, where $\boldsymbol{\phi}^T = (\phi_1, \dots, \phi_K)$.

Some bases are smooth, like the polynomial basis or the truncated Fourier basis. In such cases expressing the original data in a basis representation also have a smoothing effect.

We define the Gram matrix for a certain basis as

$$\mathbf{G} = \int_{\mathcal{T}} \boldsymbol{\phi}(t)\boldsymbol{\phi}(t)^T dt, \tag{2.3}$$

whose elements are

$$G_{ij} = \int_{\mathcal{T}} \phi_i(t)\phi_j(t)dt, \quad i,j = 1, \dots, K.$$

FPCA can be applied to both representations of a functional data sample. The theory behind both methods is the same because we are studying the data regardless of its representation. However, there are important practical differences between them, and they must be analyzed thoroughly.

The functional principal component analysis is intimately related to multivariate principal component analysis. Therefore, we start by giving an overview of the multivariate case and compare it to the functional case. In the following sections, the theoretical and practical aspects are described, following the book *Functional Data Analysis* by James Ramsay and B. W. Silverman [17].

2.1. Multivariate PCA

Abusing of the notation defined at the beginning of this chapter for the discrete representation of functional data, we use $\mathbf{X}$ to denote the sample data in the multivariate paradigm. $\mathbf{X}$ is then a $N \times M$ matrix that contains N data with M variables. Each row is an observation of the variables in this case. In multivariate statistics, linear combinations of the original variables are often used:

$$f_i = \sum_{j=1}^{M} \beta_j x_{ij} = \boldsymbol{\beta}^T \mathbf{x}_i, i = 1, ..., N.$$

Where $\boldsymbol{\beta}$ is the weight vector $(\beta_1, ..., \beta_M)^T$ and $\mathbf{x}_i$ is the vector of values $(x_{i1}, ..., x_{iM})^T$. Multivariate PCA consists in selecting the weight vector that outlines the most significant types of variation. The following steps shall be followed:

1. Find the weight vector $\boldsymbol{\beta_1} = (\beta_{11}, ..., \beta_{M1})^T$ for which the linear combinations

$$f_{i1} = \sum_{j=1}^{M} \beta_{j1} x_{ij} = \boldsymbol{\beta}_1^T \mathbf{x}_i, i = 1, ..., N \tag{2.4}$$

have the maximum

$$\frac{1}{N} \sum_i f_{i1}^2. \tag{2.5}$$

The following restriction is also imposed $\sum_{j=1}^{M} \beta_{j1}^2 = \|\boldsymbol{\beta}\|^2 = 1$ so the found weight vector is normalized.

2. The subsequent weight vectors are found in the same way, until a maximum number of M vectors. In each step $m, 2 \leq m \leq M$ we need to find a new weight vector $\boldsymbol{\beta_m}$ with values $f_{im} = \boldsymbol{\beta}_m^T \mathbf{x}_i$. The maximization criterion is the same, however, we add $m - 1$ restrictions of orthogonality

$$\sum_{j=1}^{M} \beta_{jk} \beta_{jm} = \boldsymbol{\beta}_k^T \boldsymbol{\beta}_m = 0, k < m. \tag{2.6}$$

The kth principal component of a datum x_i is $f_{ik}\beta_k$, and the values f_{ik} are called the principal component scores. The weight vectors are not unique, because inverting the sign does not change the direction of maximum variance they represent.

2.2. Definition of FPCA

There is a clear theoretical correspondence between the process described in the previous section and the process that we follow in the functional case. Recall that each $x_i(t)$ is a L^2 function defined in

a closed interval $\mathcal{T} \subset \mathbb{R}$. The vectorial product is substituted by the inner product in this space. That is, the sum 2.4 is transformed into

$$f_i = \langle \beta, x_i \rangle_2 = \int \beta(t) x_i(t) dt, i = 1, ..., N, \tag{2.7}$$

where N is the number of functions in the data sample. We omit the variable over which we integrate, assuming that it is t.

The steps to be followed are the same:

1. Find the weight function $\beta_1(t)$, but instead of maximizing 2.5 now we maximize the corresponding function

$$\frac{1}{N} \sum_{i=1}^{N} f_{i1}^2 = \frac{1}{N} \sum_{i=1}^{N} \left(\langle \beta_1, x_i \rangle_2 \right)^2, \tag{2.8}$$

with the corresponding normality restriction

$$\|\beta_1\|_2^2 = 1. \tag{2.9}$$

2. In step m we try to find the mth weight function maximizing $\frac{1}{N} \sum_{i=1}^{N} f_{im}^2 = \frac{1}{N} \sum_{i=1}^{N} (\int \beta_m x_i)^2$, with the normality restriction $\|\beta_m\|_2^2 = 1$ and $m-1$ orthogonality restrictions

$$\langle \beta_k, \beta_m \rangle_2 = 0, k < m. \tag{2.10}$$

We see that the restrictions are analogous to the multivariate restrictions of orthogonality 2.6. The main theoretical difference is that we can find an infinite number of principal components as we are in an infinite-dimensional space. Therefore, theoretically, there are no upper bounds for m. However, we will see that in practice such restrictions exist due to the fact that we can only store a finite amount of values.

The kth principal component of the datum x_i is $f_{ik}\beta_k$, and the values f_{ik} are called principal component scores, similar to the multivariate case. The obtained weight functions are called curves of the principal components. Note that this aligns with the fact that FPCA is a projection method, as the principal component scores are the projections of the original sample data into the subspace formed by the principal components.

2.3. FPCA, eigenvalues and eigenvectors

In this section, we explain how the maximization problem can be seen as an eigenvalue problem. This gives a big advantage because there are efficient methods that can be used to compute the eigenvalues and eigenvectors of a matrix, like the singular value decomposition (SVD) [20]. The differences

between the multivariate case and the functional case are again compared.

First, we suppose that the data sample is centered. Namely, the mean of the data sample is 0. If it is not the case, we can just deduct the mean from the data sample. That is, we suppose $\frac{1}{N}\sum_i x_{ij} = 0, j = 1, ..., M$ in the multivariate case or $\frac{1}{N}\sum_i x_i(t) = 0$ in the functional case.

2.3.1. The multivariate case

The criterion followed to find the first principal component 2.5 can be written as

$$\max_{\boldsymbol{\beta}^T\boldsymbol{\beta}=1} \frac{1}{N}\boldsymbol{\beta}^T\mathbf{X}\mathbf{X}\boldsymbol{\beta}, \tag{2.11}$$

as each score f_i can be written as $\mathbf{x}_i^T\boldsymbol{\beta}$. We use the matrix $\mathbf{V} = \frac{1}{N}\mathbf{X}^T\mathbf{X}$ of dimensions $M \times M$ to denote the sample variance. One may argue that $N-1$ must be used to compute the sample variance, however, it has no practical effects over the principal components.

The maximization problem can be solved finding the eigenvector of the biggest eigenvalue λ_1 of the following equation

$$\mathbf{V}\boldsymbol{\beta} = \lambda_1\boldsymbol{\beta}. \tag{2.12}$$

For the rest of the components, the subsequent eigenvectors $\beta_2, \beta_3, ...$ complies with the defined orthonormality and are solutions of the maximization equation. That is, solving this eigenequation we obtain all principal components.

2.3.2. The functional case

Lets consider the same problem in a functional setting. Instead of a covariance matrix we define the covariance operator $\mathcal{V}$ for the functions in L^2 space as

$$\mathcal{V}f(s) = \int_{\mathcal{T}} v(s,t)f(t)dt. \tag{2.13}$$

This operator is equivalent to the covariance matrix in the multivariate case. $v(s,t)$ is the covariance function defined as

$$v(s,t) = \frac{1}{N}\sum_{i=1}^{N} x_i(s)x_i(t). \tag{2.14}$$

Therefore, the problem we want to solve is

$$\int_{\mathcal{T}} v(s,t)\beta_j(t)dt = \lambda_j\beta_j(s), \quad j = 1, 2, \ldots, \tag{2.15}$$

and finally, we can express the problem as

$$\mathcal{V}\beta(s) = \lambda\beta(s). \tag{2.16}$$

The last equation is a functional eigenequation, and it is solved differently in the two different representations of functional data. The remark here is the similarity between the multivariate case and the functional case. The process is almost the same, only the involved elements are functions instead of vectors.

To solve this eigenequation a spectral decomposition is applied to the covariance operator. That is, the operator is rewritten as

$$\mathcal{V}f = \sum_{j=1}^{\infty} \lambda_j \langle f, \beta_j \rangle_2 \beta_j.$$

Where each λ_j is known as an eigenvalue and each e_j the corresponding eigenfunction. The main problem here is that in practice, we are giving a finite decomposition of the covariance operator while in theory this decomposition can be infinite. The Spectral Theorem [2, p 46] guarantees that the finite solutions we give are good approximates as they converge to the operator. We write the eigenequation as:

$$\mathcal{V}\beta_j(s) = \lambda_j\beta_j(s), \quad j = 1, 2, \ldots, \tag{2.17}$$

with $\lambda_1 \geq \lambda_2 \geq \ldots \geq 0$ and $\{\beta_j\}_{j=1}^{\infty}$ an orthonormal basis.

2.4. The importance of FPCA

There are two reasons why FPCA is important. First, it is a technique that allows users to reduce the dimensionality of the initial functional data. In the multivariate case, PCA can reduce the number of attributes of the dataset by a large percentage. In FDA, the data live in L^2, which is an infinite-dimensional space. Then the reduction in dimension is much more significant, as we go from infinite dimensions to a finite number of dimensions.

Second, the covariance operator of the data is sometimes difficult to analyze. FPCA can give a representation that is easier to understand of the said operator by applying the spectral decomposition of the operator, and representing this operator by its eigenfunctions.

Having defined the common theoretical aspects of FPCA, the specifics of implementation for each representation are discussed in the following two sections.

2.5. FPCA for discrete representation of the data

In this case, we suppose that the functions are only accessible at certain points. Therefore, the data matrix is represented using $\mathbf{X}$, which is a $N \times M$ matrix. The covariance operator defined in equation 2.13 can be approximated by the following $M \times M$ matrix

$$\mathbf{V} = \frac{1}{N}\mathbf{X}^T\mathbf{X}.$$

We need to approximate the integrals using a numeric quadrature

$$\int_{\mathcal{T}} f(t)dt = \sum_{m=1}^{M} w_m f(t_m) = \mathbf{w}^T f(\mathbf{t}) = \mathbf{w}^T\mathbf{f},$$

where $\mathbf{w}^T = (w_1, \ldots, w_M)$ is the weight vector that characterizes the numerical quadrature. For example, one simple integration rule is to use uniform weights:

$$w_j = \frac{|\mathcal{T}|}{M}, \quad j = 1, 2, \ldots, M,$$

where $|\mathcal{T}|$ is the length of the closed interval $\mathcal{T}$. We use $\mathbf{W} = \text{diag}(\mathbf{w})$ to denote the diagonal matrix of dimensions $M \times M$ whose elements in the diagonal are the elements of $\mathbf{w}$.

Using this notation, the eigenequation 2.17 for the discrete representation of functional data is

$$\mathbf{VW}\beta_j(\mathbf{t}) = \lambda_j\beta_j(\mathbf{t}); \quad j = 1, 2, \ldots M.$$

Defining $v_j(\mathbf{t}) = \mathbf{W}^{1/2}\beta_j(\mathbf{t})$, the last equation can be rewritten as

$$\mathbf{W}^{1/2}\mathbf{VW}^{1/2}v_j(\mathbf{t}) = \lambda_j v_j(\mathbf{t}); \quad j = 1, 2, \ldots M.$$

The final equation that depends on the weight matrix and the data matrix

$$\mathbf{W}^{1/2}\left(\frac{1}{N}\mathbf{X}^T\mathbf{X}\right)\mathbf{W}^{1/2}v_j(\mathbf{t}) = \lambda_j\beta_j(\mathbf{t}); \quad j = 1, 2, \ldots M. \tag{2.18}$$

Which can be solved by SVD of the following matrix

$$\frac{1}{\sqrt{N}}\mathbf{XW}^{1/2} = \mathbf{USV}^T,$$

where $\mathbf{U} = (\mathbf{u}_1, \ldots \mathbf{u_N})$ is an orthonormal matrix $N \times N$ (that is, $\mathbf{U}^T\mathbf{U} = \mathbf{UU}^T = \mathbf{I}_N$, where $\mathbf{I}_N$ is the identity matrix $N \times N$. Therefore, $\{\mathbf{u}_n\}_{n=1}^N$ is a basis in $\mathbb{R}^N$). On the other hand, $\mathbf{V} = (\mathbf{v}_1, \ldots \mathbf{v}_M)$ is an orthogonal matrix $M \times M$ (that is, $\mathbf{V}^T\mathbf{V} = \mathbf{VV}^T = \mathbf{I}_M$, where $\mathbf{I}_M$ is the identity matrix $M \times M$, and $\{\mathbf{v}_m\}_{m=1}^M$ is a basis in $\mathbb{R}^M$). Lastly, $\mathbf{S}$ is a $N \times M$ matrix, that contains a diagonal matrix in its first M rows, whose elements $s_1 \geq s_2 \geq s_{\min(N,M)}$ are called singular values. The rest of the matrix are zeroes.

The relation between those matrices is the following

$$\begin{aligned} \frac{1}{\sqrt{N}}\mathbf{X}\mathbf{W}^{1/2}\mathbf{v}_i &= s_i\mathbf{u}_i, && \forall i \leq \min(N,M), \\ \frac{1}{\sqrt{N}}\mathbf{X}\mathbf{W}^{1/2}\mathbf{v}_i &= 0 && \forall i > \min(N,M). \end{aligned}$$

Using this decomposition, the eigenequation transforms into

$$\mathbf{V}\mathbf{S}^T\mathbf{S}\mathbf{V}^T\mathbf{v}_j = \lambda_j\mathbf{v}_j; \quad j = 1,2,\ldots,M.$$

The eigenvalues are

$$\lambda_j = s_j^2; \qquad j = 1,2,\ldots,M.$$

And the corresponding eigenvectors are

$$\boldsymbol{\beta}_j = \mathbf{W}^{-1/2}\mathbf{v}_j, \quad j = 1,2,\ldots,M,$$

The eigenvectors are the eigenfunctions that we search to define the axis of projection.

We define $\mathbf{P}$ as the projection matrix of dimensions $K \times M$ containing the eigenvectors: $\mathbf{P} = (\boldsymbol{\beta}_1, ..., \boldsymbol{\beta}_M)^T$. Therefore, each row of this matrix is an eigenvector. The scores are found in the following way:

$$\mathbf{T} = \mathbf{X}\mathbf{W}\mathbf{P}^T. \tag{2.19}$$

The j-th component of the datum $x_i(t)$ is $t_{ij}\boldsymbol{\beta}_j$, where t_{ij} is the element of row i and column j of the matrix $\mathbf{T}$.

2.6. FPCA for basis representation

As explained in the beginning of this chapter, the functional data can be represented as coefficients in a certain basis $\{\phi_k\}_{k=1}^K$. We use $\boldsymbol{\xi}_i^T = (\xi_{i1}, \ldots, \xi_{iK})$ to denote the projection of $x_i(t)$ in the subspace formed by said basis, and recall that we use $\boldsymbol{\xi}$ defined in the equation 2.2 to represent the sample data. The column vector $\boldsymbol{\phi}$ is used to represent the elements of the basis. With this notation, $x_i = \boldsymbol{\xi}_i^T\boldsymbol{\phi} = \boldsymbol{\phi}^T\boldsymbol{\xi}_i$. Moreover, the letter $\mathbf{G}$ denotes the Gram matrix of this basis, whose elements are the inner products of the basis functions.

The principal component curves are expressed in another basis, $\{\varphi_k\}_{k=1}^{K'}$, that can be the same basis or a different one. In both cases, the basis may not be orthonormal, like the polynomial basis. But the principal components are orthonormal. We use the letter $\boldsymbol{\Gamma}$ to denote the Gram matrix of this basis.

The covariance function 2.14 is approximated by

$$\hat{v}(s,t) = \frac{1}{N}\boldsymbol{\phi}^T(s)\boldsymbol{\xi}^T\boldsymbol{\xi}\boldsymbol{\phi}(t).$$

If the j-th eigenfunction have the following expansion

$$\beta_j = \boldsymbol{\beta}_j^T\boldsymbol{\varphi} = \boldsymbol{\varphi}^T\boldsymbol{\beta}_j, \quad j = 1, \ldots, K',$$

then

$$\begin{aligned}\int_{\mathcal{T}} \hat{v}(s,t)\beta_j(t)dt =& \frac{1}{N}\boldsymbol{\phi}^T(s)\boldsymbol{\xi}^T\boldsymbol{\xi}\left(\int_{\mathcal{T}}\boldsymbol{\phi}(t)\boldsymbol{\varphi}^T(t)dt\right)\boldsymbol{\beta}_j \\ =& \frac{1}{N}\boldsymbol{\phi}(s)^T\boldsymbol{\xi}^T\boldsymbol{\xi}\mathbf{J}\boldsymbol{\beta}_j,\end{aligned}$$

where $\mathbf{J} = \int_{\mathcal{T}}\boldsymbol{\phi}(t)\boldsymbol{\varphi}^T(t)dt$ is a $K \times K'$ matrix whose elements are

$$J_{ij} = \int_{\mathcal{T}}\phi_i(t)\varphi_j(t)dt, \quad i = 1, \ldots, K, \quad j = 1, \ldots, K'.$$

In these terms the equation 2.15 is

$$\frac{1}{N}\boldsymbol{\phi}^T(s)\boldsymbol{\xi}^T\boldsymbol{\xi}\mathbf{J}\boldsymbol{\beta}_j = \lambda_j\boldsymbol{\varphi}(s)^T\boldsymbol{\beta}_j.$$

Multiplying by $\boldsymbol{\varphi}(s)$ on the left and integrating over $s \in \mathcal{T}$, we have the following equation

$$\frac{1}{N}\mathbf{J}^T\boldsymbol{\xi}^T\boldsymbol{\xi}\mathbf{J}\boldsymbol{\beta}_j = \lambda_j\mathbf{G}\boldsymbol{\beta}_j.$$

Rewriting the problem in terms of the Cholesky decomposition of $\mathbf{G} = \mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is a lower triangular matrix,

$$\frac{1}{N}\mathbf{J}^T\boldsymbol{\xi}^T\boldsymbol{\xi}\mathbf{J}\boldsymbol{\beta}_j = \lambda_j\mathbf{L}\mathbf{L}^T\boldsymbol{\beta}_j,$$

and defining $\mathbf{v}_j = \mathbf{L}^T\boldsymbol{\beta}_j$, we can write the eigenequation 2.15 in a symmetric form

$$\frac{1}{N}\mathbf{L}^{-1}\mathbf{J}^T\boldsymbol{\xi}^T\boldsymbol{\xi}\mathbf{J}(\mathbf{L}^{-1})^T\mathbf{v}_j = \lambda_j\mathbf{v}_j. \tag{2.20}$$

Therefore, we have to find the eigenvalues and eigenvectors of the symmetric equation

$$\left(\mathbf{J}(\mathbf{L}^{-1})^T\right)^T\left(\frac{1}{N}\boldsymbol{\xi}^T\boldsymbol{\xi}\right)\mathbf{J}(\mathbf{L}^{-1})^T = \left(\mathbf{L}^{-1}\mathbf{J}^T\right)\left(\frac{1}{N}\boldsymbol{\xi}^T\boldsymbol{\xi}\right)\left(\mathbf{L}^{-1}\mathbf{J}^T\right)^T.$$

As in the discrete case, we can apply SVD to the following matrix.

$$\frac{1}{\sqrt{N}}\boldsymbol{\xi}\mathbf{J}(\mathbf{L}^{-1})^T.$$

If the basis for the eigenfunctions and the original data is the same, then we can simplify the equation

because $\mathbf{J} = \mathbf{G} = \mathbf{L}\mathbf{L}^T$:

$$\mathbf{L}^T \left(\frac{1}{N} \boldsymbol{\xi}^T \boldsymbol{\xi} \right) \mathbf{L}\mathbf{v}_j = \lambda_j \mathbf{v}_j.$$

Finally, assuming that the eigenvalues are given in decreasing order, $\lambda_1 \geq \lambda_2, \ldots, \lambda'_K$, the eigenfunctions are

$$\boldsymbol{\beta}_j = (\mathbf{L}^{-1})^T \mathbf{v}_j, \quad j = 1, 2, \ldots, K',$$

and the principal component scores are obtained integrating

$$t_{ij} = \int \beta_j x_i = \int \beta_j(s) x_i(s) ds, i = 1, ..., N, j = 1, ..., K', \tag{2.21}$$

which can be further expanded as

$$t_{ij} = \langle \beta_j, x_i \rangle_2 = \int \boldsymbol{\beta}_j \boldsymbol{\varphi}^T \boldsymbol{\xi}_i \boldsymbol{\phi}^T = \boldsymbol{\beta}_j \left(\int \boldsymbol{\varphi}^T \boldsymbol{\phi} \right) \boldsymbol{\xi}_i^T = \boldsymbol{\beta}_j \mathbf{J}^T \boldsymbol{\xi}_i^T = \boldsymbol{\xi}_i \mathbf{J} \boldsymbol{\beta}_j^T.$$

The j-th principal component of the datum $x_i(t)$ is $t_{ij}\beta_j$. The symbol $\mathbf{T}$ is used again to define the matrix whose elements are the principal component scores.

We observe that in both representations there are restrictions over the number of principal components that can be found. In the discrete case, the limit is M, and in the basis case, the number is K'. We recall that the Spectral Theorem [2, p 46] guarantees that this works properly for an infinite-dimensional operator.

Note that in equation 2.20 we need to invert the matrix $\mathbf{L}$ in order to multiply it with other matrices. In practice, this is equivalent to solving a linear matrix equation of the form $\mathbf{L}\mathbf{A} = \mathbf{B}$, where $\mathbf{B}$ is the rest of the matrix to which we want to multiply $\mathbf{L}^{-1}$ and $\mathbf{A}$ is the unknown matrix. If we solve this equation we obtain $\mathbf{A} = \mathbf{L}^{-1}\mathbf{B}$, which is the desired result. Solving this problem as a linear matrix equation has two advantages: first, it is more stable because we do not invert a matrix in the process, and matrix inversion can induce stability problems. Near zero entries in the matrix may result in large errors when inverted. Second, it is more efficient, employing less time for the same task.

2.7. Regularized FPCA

In most cases when we obtain the curves of principal components these can present a big amount of curvature. This is due to the irregularities present in the data sample, that are frequently caused by noise. There are methods available in the library `scikit-fda` that can smooth the original dataset, which in turn results in smoother principal components. However, smoothing can be applied directly to the principal components during the process of FPCA, which we refer to as regularization. We use D^k to denote the derivative of order k.

2.7.1. Roughness estimation

One way to estimate the amount of irregularity present in curves is the amount of curvature. We approximate the curvature of a function $x(t)$ as $D^2x(t)$. The penalty of a function $x(t)$ is then defined as

$$p(x;2) = \int [D^2x(t)]^2 dt,$$

where the curvature is squared to eliminate the sign. This is because of either negative or positive, if the absolute value is sizeable, then the amount of variation in the curve will be big. We expect that a function with a large amount of variability to have a bigger value of penalty. A straight line have curvature 0, therefore, its penalty is 0.

The roughness of a curve can be defined more generally, allowing k order derivatives, where $k > 0$ is an integer.

$$p(x;k) = \int [D^kx(t)]^2 dt, \tag{2.22}$$

and it is also possible to use linear combinations of $p(x;k)$ to penalize different order derivatives at the same time. In fact, it is possible to penalize any linear operator using Tikhonov regularization.

2.7.2. Finding the first regularized principal component

We recall that in FPCA the goal is to maximize the sample variance represented by the principal component β with the restriction of $\|\beta\| = 1$. This is equivalent to maximizing the equation 2.8. The goal in regularization is to also minimize $PEN_2(\xi)$. The penalized sample variance is defined as

$$psv(\beta) = \frac{\frac{1}{N}\sum_{i=1}^{N}(\langle\beta_1, x_i\rangle)^2}{\|\beta\| + \lambda \times PEN_2(\beta)}, \tag{2.23}$$

where $\lambda \geq 0$ is a parameter that controls the importance of the penalty term. A bigger λ implied larger penalization. the psv makes explicit both goals.

Now we find the first principal component under the same restriction of $\|\beta\| = 1$. Observe that when $\lambda = 0$, we revert back to the original case because $\|\beta\| = 1$. When $\lambda \to \infty$, the curve of the principal component is forced to be a straight line.

2.7.3. Estimation of subsequent penalized principal components

The estimation of subsequent principal components is done maximizing $psv(\beta_j)$, with additional restrictions:

1. The usual restriction $\|\beta_j\| = 1$.

2. A modified form of orthogonality:

$$\int \beta_i(s)\beta_j(s)ds + \lambda \int D^2\beta_i(s)D^2\beta_j(s)ds = 0, i = 1, ..., j-1 \tag{2.24}$$

The second condition defined a new inner product, that depends on the regularization parameter λ.

$$\int \beta_i(s)\beta_j(s)ds + \lambda \int D^2\beta_i(s)D^2\beta_j(s)ds = 0. \tag{2.25}$$

2.7.4. Regularized FPCA for discrete representation

The penalty matrix for discrete FPCA can be computed as the paper written by Krämer, Nicole and Boulesteix, Anne-Laure and Tutz, Gerhard [12] presents.

Recall that the data matrix is represented using $\mathbf{X}$, of dimensions $N \times M$. M indicates the number of points of discretization. Let $\mathbf{t} = (t_1, ..., t_M)^T$ be the vector with the sample points, and let $\boldsymbol{\beta}$ be a random curve. We have to compute the second order differences in order to penalize the curvature the the functions. The penalty for $\boldsymbol{\beta}$ is given as

$$p(\beta; 2) = \lambda\boldsymbol{\beta}^T(\mathbf{D}_{M-2}\mathbf{D}_{M-1})^T(\mathbf{D}_{M-2}\mathbf{D}_{M-1})\boldsymbol{\beta} = \lambda\boldsymbol{\beta}^T\mathbf{P}\boldsymbol{\beta} \tag{2.26}$$

Here, the matrix $\mathbf{D}_K$ is a $K \times (K+1)$ matrix

$$\mathbf{D}_K = \begin{pmatrix} h_1 & -h_1 & 0 & ... & 0 & 0 \\ 0 & h_2 & -h_2 & ... & 0 & 0 \\ \vdots & & & & & \\ 0 & 0 & 0 & ... & h_K & -h_K \end{pmatrix}, h_j = \frac{1}{t_j - t_{j+1}}$$

This can be easily extended to compute higher order penalty matrixes, and linear combinations of them. Let k be the order of the derivative, we can define the penalty as

$$p(\beta; k) = \lambda\boldsymbol{\beta}^T(\mathbf{D}_{M-k}...\mathbf{D}_{M-2}\mathbf{D}_{M-1})^T(\mathbf{D}_{M-k}...\mathbf{D}_{M-2}\mathbf{D}_{M-1})\boldsymbol{\beta} \tag{2.27}$$

then the equation 2.26 is a particular case of this equation, with $k = 2$.

We can also compute the penalty matrix for linear combinations of derivative operators of different orders. We just have to compute the penalty matrix for each one of them, and then add the matrixes together using the coefficients of the linear combination.

In practice, this is combined with discrete FPCA in the following way. The eigenequation that we need to solve is equation 2.18. Adding the penalty term results in the following equation

$$\frac{\mathbf{W}^{1/2}\left(\mathbf{X}^T\mathbf{X}\right)\mathbf{W}^{1/2}v_j(\mathbf{t})}{N(\mathbf{I}_M+\mathbf{P}_k)^T(\mathbf{I}_M+\mathbf{P}_k)}=\lambda_j\beta_j(\mathbf{t});\quad j=1,2,\dots M, \tag{2.28}$$

which can be solved using eigen-analysis methos or SVD on

$$\frac{\mathbf{X}\mathbf{W}^{1/2}}{\sqrt{N}(\mathbf{I}_M+\mathbf{P}_k)}$$

2.7.5. Regularized FPCA for basis representation

In this case, the penalty matrix $\mathbf{P}$ is computed as the Gram matrix of the basis functions' derivatives. For each basis the derivatives are computed analytically.

To apply the penalty to FPCA, just have to add the penalty matrix $\mathbf{P}$ to the Gram matrix $\mathbf{G}$ of the basis in which the functional data is expressed. Recall that in equation 2.20 the matrix $\mathbf{L}$ is a lower triangular matrix that results from the Cholesky decomposition of $\mathbf{G}$. That is, $\mathbf{G}=\mathbf{L}\mathbf{L}^T$. This decomposition is now done after adding the penalty term. The Cholesky decomposition is now $\mathbf{G}+\lambda\mathbf{P}_k=\mathbf{L}\mathbf{L}^T$. Therefore, the penalty term is implicitly included in the matrix $\mathbf{L}$.

2.7.6. Finding the regularization parameter

It is difficult to determine the regularization parameter λ that we should use to smooth the principal component curves. Because a poorly chosen parameter can result in over-smoothing or under-smoothing. Here a leave one out cross validation method is presented [17, p 178].

First, the process of FPCA for functional data in basis representation is summarized. Recall that the matrix $\boldsymbol{\xi}$ of dimensions $N\times K$ represent the functional data in a basis representation, where K is the number of functions in the chosen basis. An essential element in FPCA is the projection matrix $\mathbf{P}$ that projects the original data into the subspace formed by the k first principal components. Projecting in the functional contexts is equivalent to computing the inner product of the original data and the principal component curves. The matrix $\mathbf{P}$ are the coefficients of the principal component curves in the chosen basis. Then, the dimensions of this matrix are $k\times K$. It depends on both the regularization parameter and the number of principal components, and this relation is expressed as $\mathbf{P}(\lambda,k)$. Note that $k\leq K$, as the principal component curves are orthonormal linear combinations of basis functions. If there are more principal component curves than basis functions, these cannot be orthonormal, resulting in a contradiction.

The goal is to obtain the principal component scores that results from projecting the original data.

We use $\mathbf{T}$, of dimensions $N \times k$, to denote the principal component scores as in the sections 2.5 and 2.6. FPCA can be summarized as follows

$$\boldsymbol{\xi} \xrightarrow[\mathbf{P}(\lambda,k)]{FPCA} \mathbf{T}(\lambda,k),\ \mathbf{T}(\lambda,k) = \boldsymbol{\xi}\mathbf{P}(\lambda,k)^T. \tag{2.29}$$

Here, the product between the matrix $\boldsymbol{\xi}$ and $\mathbf{P}$ symbolize the inner product between their elements. That is, the element t_{ij} of the matrix $\mathbf{T}$ is $t_{ij} = \langle \xi_i, \phi_j \rangle_2$, with ϕ_j the j-th principal component curve, with coefficients p_j, the j-th row of the matrix $\mathbf{P}$.

Let $\boldsymbol{\xi}^{[\setminus i]}$ be the sample data without the i-th datum. This is a $(N-1) \times K$ matrix. As the sample data changed, the projection $\mathbf{P}$ is also affected, and we use $\mathbf{P}^{[\setminus i]}(\lambda, k)$ to show the difference. The equation 2.29 is now:

$$\boldsymbol{\xi}^{[\setminus i]} \xrightarrow[\mathbf{P}^{[\setminus i]}(\lambda,k)]{FPCA} \mathbf{T}^{[\setminus i]}(\lambda,k),\ \mathbf{T}^{[\setminus i]}(\lambda,k) = \boldsymbol{\xi}\mathbf{P}^{[\setminus i]}(\lambda,k)^T.$$

The principal component scores of the datum that was left out can be computed in the following way:

$$\mathbf{t}_i^{[\setminus i]}(\lambda,k) = \mathbf{P}^{[\setminus i]}(\lambda,k)\boldsymbol{\xi}_i.$$

Let $\{\phi_j^{[\setminus i]}(\lambda)\}_{j=1}^k$ be the first k obtained principal component curves, and let

$$\boldsymbol{\phi}^{[\setminus i]}(\lambda,k) = (\phi_1^{[\setminus i]}(\lambda), \phi_2^{[\setminus i]}(\lambda), ..., \phi_k^{[\setminus i]}(\lambda))^T$$

be the vector with the principal component curves. Then, the projection of the datum that was left out is $(\mathbf{t}_i^{[\setminus i]}(\lambda,k))^T \boldsymbol{\phi}^{[\setminus i]}(k)$. The error of this projection is defined as the L^2 norm of its difference with the datum:

$$e_i^{[\setminus i]}(\lambda,k) = \|\boldsymbol{\xi}_i^T\boldsymbol{\varphi} - \mathbf{t}_i^{[\setminus i]}(\lambda,k)^T\boldsymbol{\phi}^{[\setminus i]}(\lambda,k)\|_{L_2}^2. \tag{2.30}$$

This error is computed for each $k, 1 \leq k \leq K$ to make the final error independent of the number of principal components found:

$$e_{loo-cv}(\lambda,k) = \frac{1}{N}\sum_{i=1}^{N} e_i^{[\setminus i]}(\lambda,k), \tag{2.31}$$

and the error for a certain parameter λ is defined as

$$e_{loo-cv}(\lambda) = \sum_{k=1}^{K} e_{loo-cv}(\lambda,k). \tag{2.32}$$

The objective is to find a λ that minimizes this error. Usually, this is computed for λ in a logarithmic scale.

2.8. Depth based median and trimmed mean

In addition to FPCA, other concepts the depth-based median and the trimmed means for functional data have also been studied. The notion of depth for a curve is related to its centrality within a given group of curves. A curve with a higher depth value is more centered. In the article *Trimmed means for functional data* [7] some depth measures are discussed. If we rank the data according to its depth value, then the first data, with the highest depth value, is denominated as the depth-based median.

Let N be the number of data in a given sample, and α be the proportion of data we want to cut from the data sample, then the trimmed mean for functional data is defined as the mean of the $N - N\alpha$ curves with higher depth values. The objective is to exclude the least centered curves (outliers) from the dataset.

In the library `scikit-fda` [8] there are several depth measures already implemented: band depth, modified band depth, and Fraiman and Muniz Depth. One example using a real-world dataset of trimmed mean and depth based median is presented in Appendix A.

3

Analysis, design and development

In the previous chapter the process of FPCA is discussed in detail for both discrete representation and basis representation. In this chapter the implementation process of the described functionalities in the package `scikit-fda` [8] is shown.

As the software is open-source, it is maintained by multiple developers throughout its lifespan, so specific requirements that ensure code robustness and readability are presented in the analysis section. The overall design of the library that was already established is shown in the design section, with an emphasis in the modules closely related to FPCA. Several tools were used to ensure code quality: documentation, version control, unit testing, and continuous integration.

3.1. Analysis

The package is an open-source project that started in 2018 as part of the degree work of Miguel Carbajo [1], and is now being developed by the Machine Leaning Group (GAA, Grupo de Aprendizaje Automático) at the Autonomous University of Madrid (UAM, Universidad Autónoma de Madrid). The link to the repository is `https://github.com/GAA-UAM/scikit-fda`. The package currently supports Python 3.6 and 3.7.

The requisites established in 2018 are incorporated to this work:

- The developed software has to be in Python.
- The coding standards and documentation standards established in PEP 8 y PEP 257 must be followed.
- The project must be open-source.
- As the package is intended for the general audience, the documentation must be as thorough as possible.
- The software has to be scalable, establishing a mechanism that allows easy contributions to the software.

- The project must include extensive unit tests and continuous integration mechanisms.
- All algorithms must be implemented prioritizing efficiency. Therefore, methods already implemented in `numpy` [14], `scipy` [11] and `scikit-learn` [15] should be preferred as they are coded in efficient low-level languages like C or Fortran.

New requisites have been added in the development of the package:

- The software should be cross-platform, therefore, the unit tests must support the main operating systems: Linux, MacOs and Windows.
- The API should be similar to `numpy` [14], `SciPy` [11] and `scikit-learn` [15].
- The documentation should also contain examples showing the implemented functionalities.

3.2. Design

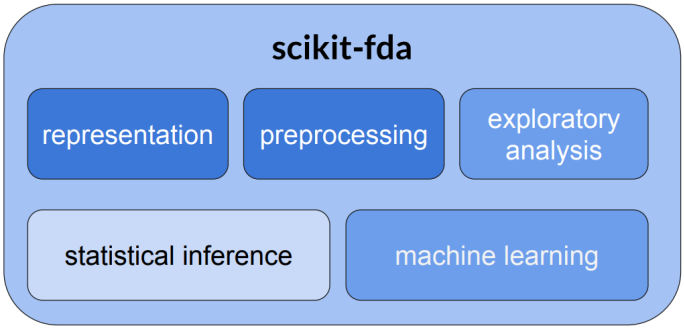


Figure 3.1: Overall structure of scikit-fda.

The structure of the package has changed considerably since 2018. Only the latest design is discussed in this section, with an emphasis on the modules representation and preprocessing. A hierarchical structure is followed, with the main modules shown in Figure 3.1, which was presented in the III International Workshop on Advances in Functional Data Analysis [16]. Darker colors indicate a more complete module. In general, the package has been developed using an object-oriented paradigm.

3.2.1. Representation

This module includes functionalities regarding the two principal functional data representations discussed in Chapter 2. Two classes, FDataGrid and FDataBasis, implement the discrete representation and the basis representation of functional data respectively. These classes inherit the abstract class

FData, which contains the common attributes and methods used in both representations, like the number of data in a certain sample. For the basis representation, extra classes are created to implement the Constant, Fourier, Monomial, and BSplines basis.

3.2.2. Preprocessing

This module contains functionalities that preprocess the data prior to analysis. There are three main submodules, Smoothing, Registration, and Dimensionality Reduction. The Smoothing module contains classes that implement various methods capable of reducing the irregularities present in the data, reducing data noise. The Registration module is in charge of aligning the data as the functional data domain may be misaligned. It also contains alignment based on features of the data, such as maxima.

FPCA is implemented under the submodule Dimensionality Reduction, as its main goal is to reduce the dimension of the original data. It is further classified as a technique based on projection, due to the fact it projects the original data onto a subspace formed by the principal components. Currently, this is the only class implemented inside the submodule. However, other dimensionality reduction methods are considered for future work.

3.2.3. Exploratory analysis

This module contains techniques that summarize, interpret, and visualize functional data as well as the information obtained from processing the data using other modules. It contains three submodules: Visualization, Depth Measures, and Outlier Detection. In the visualization module, a method that helps with the understanding of the directions of maximum variance the principal components represent is implemented. Examples with how this method works can be found in Chapter 4.

The depth measures mentioned in the section 2.8 is inside the module Depth Measures. The trimmed mean and depth-based median are considered as internal functions, and are located inside the directory `skfda/exploratory/stats/`.

3.2.4. Machine Learning

The three submodules inside this module are Classification, Clustering, and Regression. In the AEMET weather example, 4.2 in Chapter 4 the functionalities from the module Clustering were used in conjunction with FPCA to analyze the data.

3.2.5. Inference and other modules

The inference module is the least mature module, containing functional ANOVA. There are two other auxiliary modules in the package: a Datasets module that is in charge of fetching real datasets as well as generating synthetic datasets; a Miscellaneous module that contains miscellaneous functions and objects.

3.2.6. The FPCA class

The class FPCA is implemented in the directory `skfda/preprocessing/dim_reduction/projection`. It can accept both FDataBasis and FDataGrid objects, and processes the passed data according to the implementation details discussed in Sections 2.5 and 2.6.

The class has 5 constructor parameters and 3 instance attributes. The constructor parameters are:

1. `n_components`: number of components we want to obtain.
2. `centering`: indicates if we should center the data.
3. `regularization`: specifies how the components should be penalized.
4. `components_basis`: only for FDataBasis objects, indicates the basis of the principal components. If it is not specified then the same basis as the passed FDataBasis object will be used.
5. `weights`: a vector that specifies the numerical quadrature that should be used for FDataGrid objects. Refer to the matrix $\boldsymbol{W}$ in Section 2.5.

The 3 instance attributes are `components_`, which contains the principal component curves in discrete representation or basis representation; `explained_variance_`, the amount of variance explained by each component; and `explained_variance_ratio_`, the percentage of total variance explained by each component.

The class implements `scikit-learn` classes `BaseEstimator` and `TransformerMixin`, in order to conform to `scikit-learn` estimator standards. Therefore, the class contains the following mandatory methods: `fit`, `transform`, and `fit_transform`. Because the FPCA process is different for FDataBasis and FDataGrid objects, four additional internal methods are added, two for each type of object. Those methods are `_fit_grid`, `_transform_grid`, `_fit_basis`, and `_transform_basis`. All these methods accepts two arguments, X (a FDataGrid or FDataBasis object, depending on the suffix) and y (only present for convention). In `fit` methods the principal component curves are found, whereas in `transform` methods the scores are returned.

In `scikit-learn` we can use pipelines to assemble several steps that we want to apply to the data sequentially. For example, we can integrate preprocessing steps with classification steps. This way, we can simplify the workflow and enforce the order of steps. Because the class conforms to `scikit-learn` standards, we can integrate it within a `scikit-learn` pipeline. This is shown in

Code 3.1: Illustration of `scikit-learn` Pipeline usage with the FPCA class.

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from skfda.preprocessing.dim_reduction.projection import FPCA
from skfda.datasets import fetch_growth
berkeley = fetch_growth()
X = berkeley['data']
y = berkeley['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
pipeline = Pipeline([
    ('fpca',FPCA(n_components=2)),
    ('clf', LogisticRegression())
```

Code 3.1, where we use FPCA as a preprocessing step to logistic regression.

3.3. Coding

The most important aspect considered in the coding process was to strictly follow established standards. This is because of the participation of multiple developers, who have to integrate and review the work of each other. Furthermore, the code is open-source, and other community developers may contribute to the library. Following strict standards not only makes the code more readable but also simplifies the integration process.

Two Python Enhancement Proposals (PEP) are considered for this work. PEP 8, *Style Guide for Python Code*, which includes the conventions followed in the code, and PEP 257, *Doctring Conventions*, which establishes how the code should be documented.

During the coding process, efficiency was a top priority. FPCA for both representations of functional data can be reduced to a standard multivariate PCA, and then the method *sklearn.decomposition.PCA* us used to finish the process. This method is highly optimized and maintained by other professionals. Also, matrix stability problems caused by matrix inversions are averted, as explained in the Section 2.6.

3.4. Documentation and unit tests

As established earlier in the requirements, the documentation must be thorough and extensive. There are two parts of the documentation, the first one is related to the PEP 257 standard. Docstrings following said standard can be used to generate readable html pages or pdfs, where details of the class are shown. In this case, the documentation tool used is called Sphinx. Additional explanation about the module is also added in external files using this tool, which adds extra content to the generated pages.

The latest docs are available in `https://fda.readthedocs.io/en/latest/`. The documentation related to the implemented FPCA module is included in Appendix C. Furthermore, documenting the code using docstrings also allows for a type of test called doctests, which are embedded tests inside the code. These tests are elementary tests to illustrate how the class works and guarantee that the documentation is updated with code changes.

The second part of the documentation consists of an example illustrating how the FPCA class works. The example is generated from the source file *examples/plot_fpca.py*. The location of the generated example is `https://fda.readthedocs.io/en/latest/auto_examples/plot_fpca.html`. In this example, FPCA is used to analyze a real-world dataset. The main goal is to illustrate how FPCA works and how it is used to reduce the dimensionality of the data.

Apart from the doctests, unit tests were created to ensure that the FPCA class works correctly and that the obtained results align with other R libraries that also implement FPCA. The unit test file is located in *tests/test_fpca.py* inside the root directory of the package. In this file, tests were created to ensure that the arguments are checked. Moreover, the results obtained from FPCA for the discrete representation of a certain dataset is compared to the library *fda.usc* [6], and the results obtained from FPCA for basis representation of the same dataset is compared to the results from the library *fda* [18].

3.5. Development, version control and continuous integration

During the development process, weekly meetings, both online or offline, were held to ensure the project progress. During these weekly meetings, problems regarding implementation details were discussed and results in the form of reports were shown. Theoretical doubts were also discussed among the members of the development team.

The version control system used in this project is git as the project in an open-source project on Github. This platform offers tools for reviewing processes and integration of individual contributions. The branching model followed in this project is called Gitflow [4], which ensures a fluid integration of the implemented code into the package.

The model is organized around two main branches, *develop* and *master*. The branch *master* should always be in a production-ready state, with polished and complete functionalities. The branch *develop* is where the developers integrate their work into for the next stable release. Supporting the main branches other smaller branches may be created, of three different types:

1. *feature* branches. These will be the most common branches, in each branch a complete feature with shall be implemented.

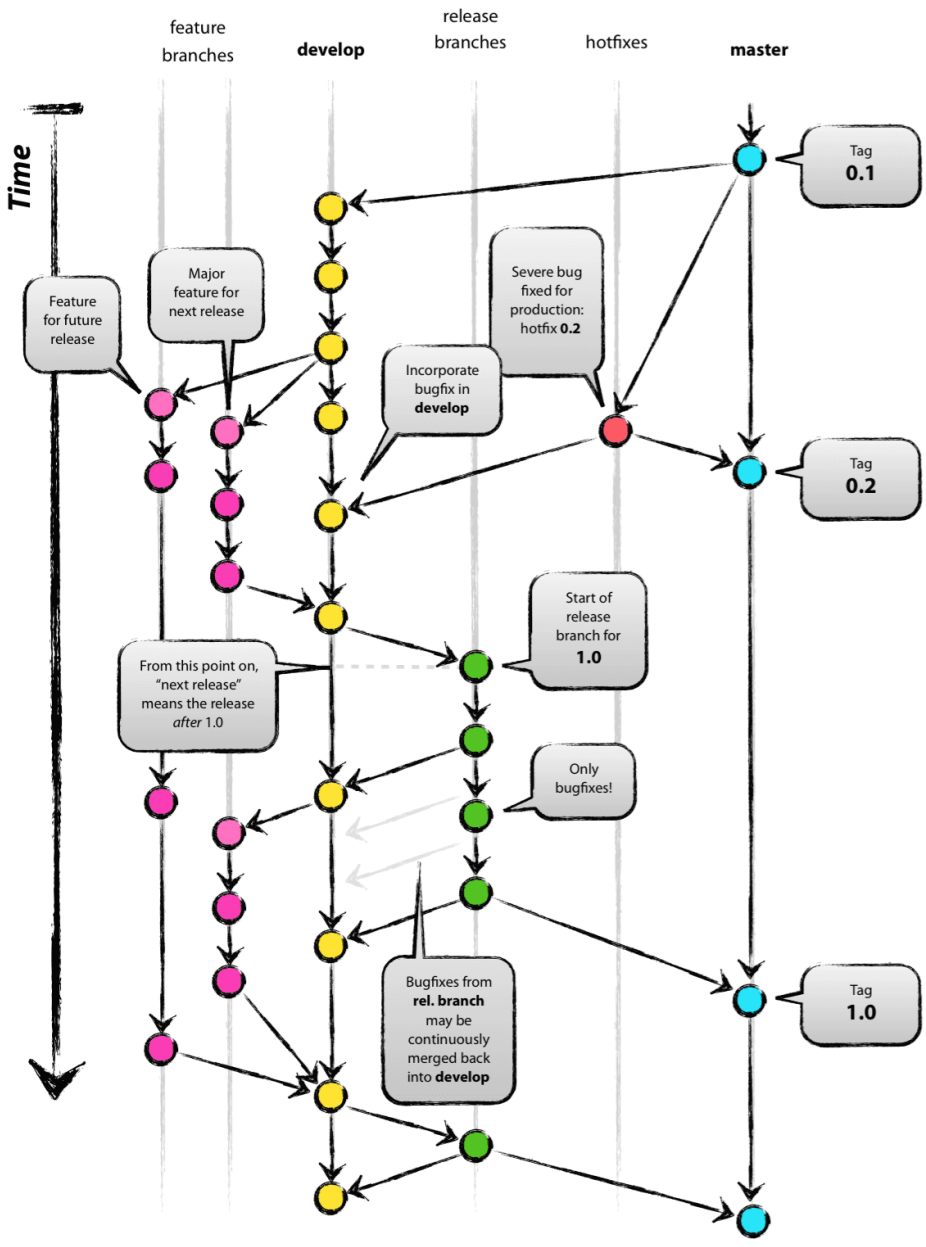


Figure 3.2: Gitflow model.

2. *hotfix* branches. These aim to address urgent bugs in the software.
3. *release* branches. These aim to prepare the *develop* branch to create a stable release in the *master* branch.

The *feature* and *hotfix* branches should merge into *develop*. The *release* and *hotfix* branches should merge into *master*. We can visualize this branching model in Figure 3.2.

When a developer wants to merge a branch into the branch *develop*, a pull request is created where other developers review the peer code and comment on the code quality as well as correctness. After peer approval, the auxiliary branch can then be merged. This further improves code quality as only reviewed code is integrated.

Github can also connect with a third-party continuous integration tool called Travis CI, which is used in the project to ensure continuous integration. Travis CI compiles the library, and runs all the tests including doc tests and unit tests each time a commit is made. This way, we ensure continuous integration as no change can be integrated unless all the tests are passed.

4

Results

In this chapter, the method FPCA is applied to three different datasets. The objective is to show the practical usage of FPCA, and how it can be used to analyze everyday data. The integration with *scikit-learn* standards is also shown, as the method can be used in a pipeline with *scikit-learn* modules. The notebook used to obtain these figures is included in Appendix B.

4.1. Berkeley Growth Study

This data sample is obtained from a growth study [21] conducted by the University of California, Berkeley. The data sample contains 93 growth curves from age 1 to age 18 of children living in California. From which, 39 are males and 54 are females.

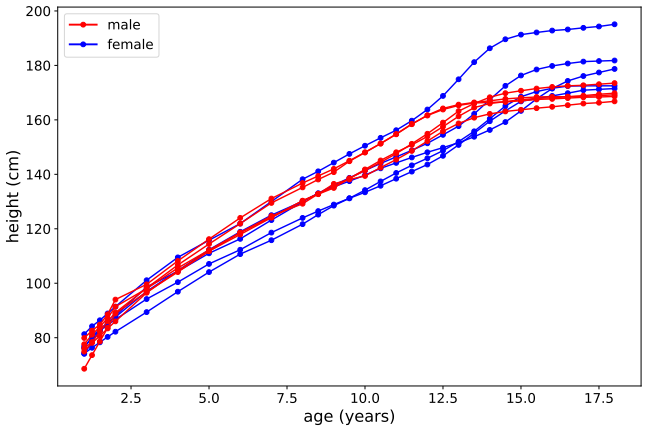


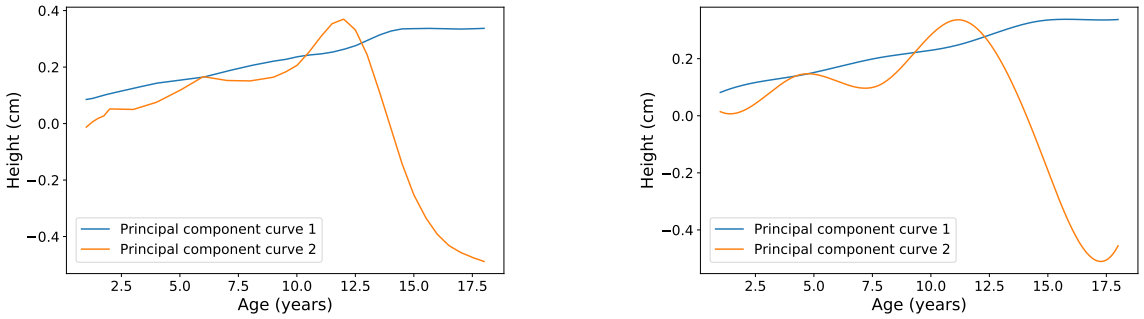
Figure 4.1: Ten growth curves from the Berkeley Growth Study.

In Figure 4.1 ten curves of the sample data are displayed. The sample points are not equidistant, with measurements made every year from age 2 to 8, and every half year in the rest of the time interval.

The objective of this sample is to show how principal component curves can aid us to understand better the sample data. Moreover, we show how the dimensionality reduction works in FPCA and how

it can help with classification problems. First, we choose an appropriate basis for this data sample. Then, we apply FPCA to the sample data in both discrete and basis representations and we obtain the principal component curves and the principal component scores. Then we plot the relevant graphs and analyze them. For details of computation refer to sections 2.5 and 2.6.

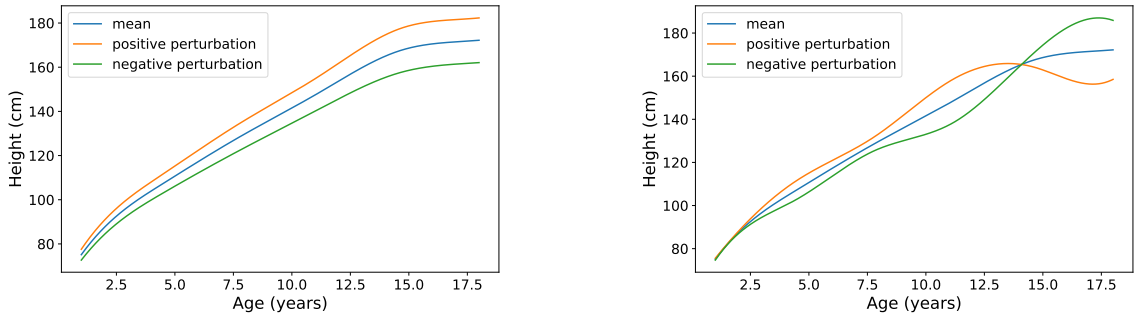
The chosen basis for the data is BSplines as the data can be reasonably approximated using piecewise polynomials. The chosen number of functions in the basis is 8.



(a) In grid representation

(b) In basis representation

Figure 4.2: The first two principal component curves of the Berkeley Growth Study, in both representations



(a) First principal component

(b) Second principal component

Figure 4.3: The perturbations over mean for the first two principal components of the Berkeley Growth Study

In Figure 4.2 the first two principal component curves are shown. We observe that the principal components in basis representation and in discrete representation are very similar. This effect should happen when the chosen basis is appropriate, as the process is theoretically the same for the data in either representation. Recall that the principal components show the most important modes of variation in the data. However, by observing the curves the information obtained is limited. Recall that the principal components conserve the most sample data variance.

In Figure 4.3 we plot the perturbations over the mean of the two first principal components. By definition, variance represents the differences of the data with respect to the mean, then it is sensible to plot the principal components as variations of the mean. In this figure, we plot the mean curve with two extra curves: adding a multiple of the principal component curve to the mean curve and subtracting a multiple of the principal component curve from the mean curve. A data curve with a positive principal component score is expected to behave similarly to the positive perturbation curve, and a low principal component score indicates the contrary. It becomes clear that if a data curve has a positive score in the first principal component, then it is overall higher than the mean curve. A negative score indicates the contrary. The first principal component also indicates that the variations of the sample data with respect to the mean increase as the children grow older. The first principal component can be summarized as an overall change in the mean height.

We can observe that the second component is indicative of the differences between early or late puberty. This is because a positive second component score indicates that the growth spurt of a child starts at an early age. Note that the positive perturbation curve is higher than the mean curve at the beginning of the interval, and then decelerates near the knot located at 14 years.

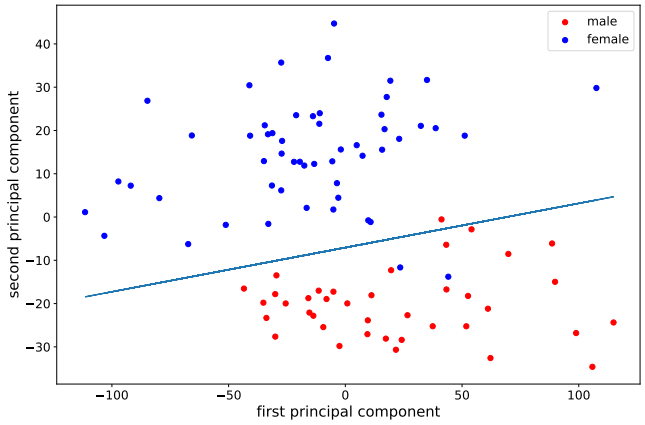


Figure 4.4: Dispersion diagram of the first two principal components of the Berkeley Growth Study, with logistic regression decision boundary.

In Figure 4.4 the dispersion diagram for the first two principal components is shown. Note that in this diagram the original data, of infinite dimensions, are shown in two dimensions.

We can see that the dots representing boys and girls are nicely grouped. Generally speaking, boys (blue dots) tend to have a higher first principal component score and a lower second principal component score. This means that the boys are generally higher than the girls and enter puberty later than girls, respectively. The girls have a lower first score and a higher second score, which shows that they differ from the boys by being smaller and having the growth spurt sooner than boys. These conclusions are coherent to the general notion we have regarding children's growth.

If we want to categorize the curves as boys or girls, then it is difficult if we were using the original data. After applying the FPCA process, this problem can be solved with high precision using only two attributes. For example, logistic regression [10] implemented in *scikit-learn* [15] can classify the curves with a precision value of $96,77\,\%$. The decision boundary, linear in logistic regression, is shown in the same Figure 4.4 as a blue line.

4.2. AEMET weather data

This data sample contains the daily temperature, precipitation and wind intensity of 73 different stations in Spain. The stations are from 41 provinces. The data was obtained from the R library *fda.usc* [6] and was elaborated by the authors of the library using data obtained from the Agencia Estatal de Meteorología de España (AEMET). For this example we only use the temperature data, which contains the average for the period 1980-2009 of daily temperature at each station. The main objective is to analyze a data sample from real life using FPCA.

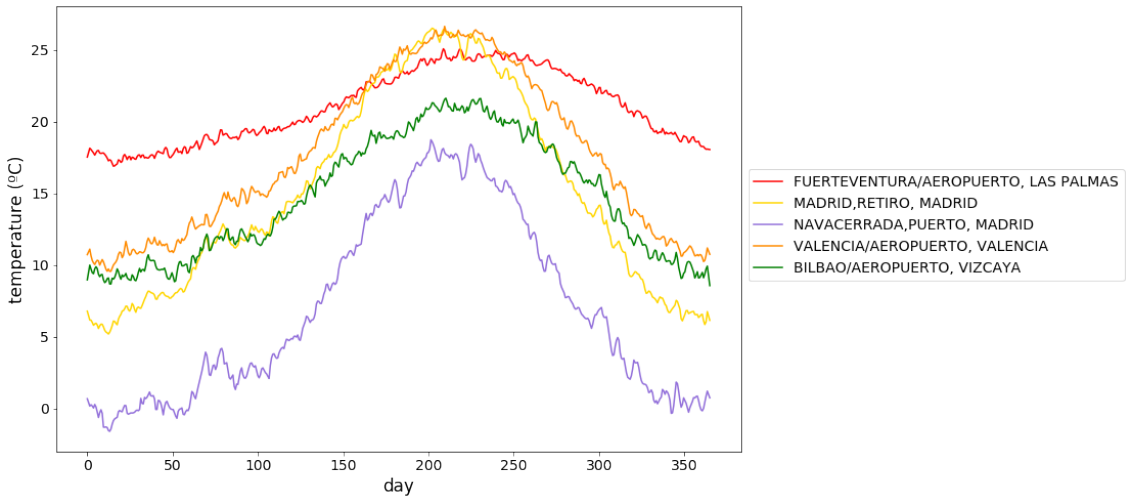
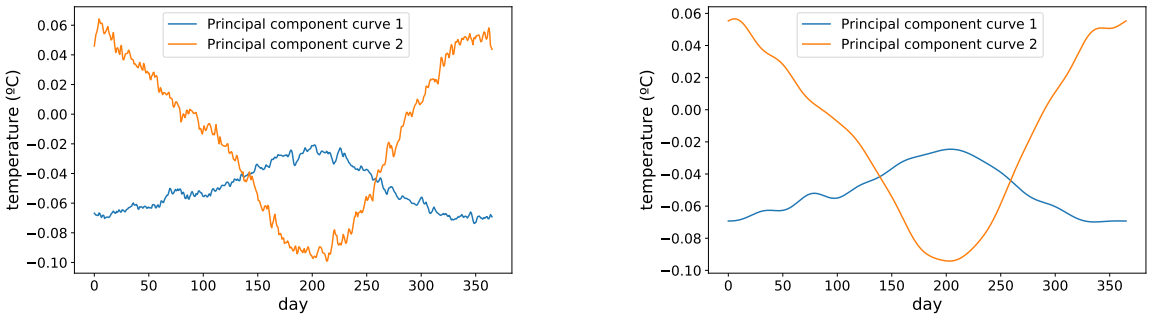


Figure 4.5: Five curves from the AEMET temperature data, from 5 chosen stations.

In Figure 4.5 five daily temperature curves from five different stations are shown. These stations represent the five different climates in Spain, using as reference [3] and [13]. The station in red, located in Las Palmas, can be easily distinguished because of its subtropical climate, which softens and elevates the curve above others. The yellow curve is located in the center of Madrid, with a continental climate. The purple station is in a mountain range in the province of Madrid, and has a much lower average temperature due to its alpine climate. Valencia (in orange) has a rather characteristic mediterranean climate, with higher temperatures in winter. And lastly, in green we have Vizcaya with an oceanic climate, which also softens the curve but is not as warm as in Las Palmas.

We compute the first two principal components for both representations of functional data, which is shown in Figure 4.6. The chosen basis is the Fourier basis with period 365, as the data is periodic

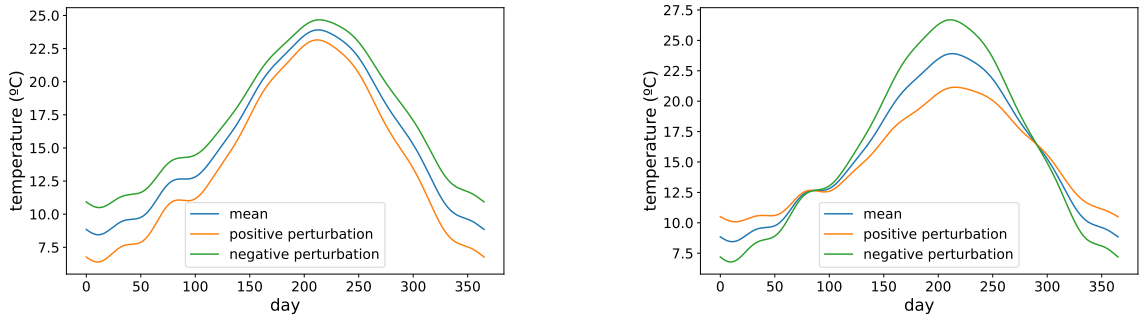


(a) First principal component

(b) Second principal component

Figure 4.6: First two principal components of the AEMET weather dataset.

with period 365. The number of basis functions is 21. The perturbation diagram and the dispersion diagram for the first two principal components are shown to illustrate the modes of variation of these components. In this case, we use a clustering algorithm to divide the data into 5 clusters, one for each described climate, using the module as part of the degree thesis [9] of Amanda Hernando *skfda.ml.clustering.KMeans*.



(a) First principal component

(b) Second principal component

Figure 4.7: Perturbations of the mean for the first two principal components for the AEMET temperature data.

The first principal component is again an overall change of on the mean, as in the Berkeley Growth Study. Note that a curve with a high first principal component score has a lower overall mean. Moreover, the variations in winter temperatures are higher than in summer temperatures. The second principal component is much more interesting, as we can interpret it as a maritime influence on the climate. This is more notable in the subtropical and oceanic weather, where summer and winter temperatures differ less. A high positive value in the second component indicates this phenomenon.

In Figure 4.8 we visualize the dispersion diagram for this dataset using the first two principal component scores. Different colors are used to distinguish the clusters. By analyzing the component scores,

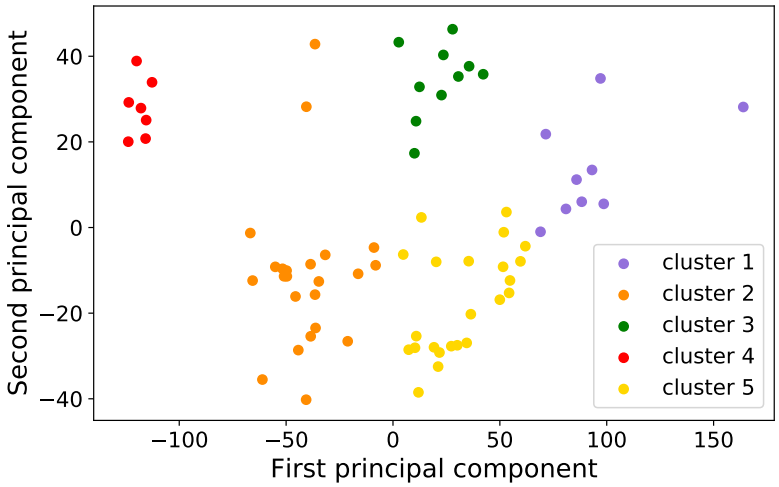


Figure 4.8: Dispersion diagram for the AEMET temperature data, dividided into 5 clusters.

we can assign a climate to each cluster. In conjunction with Figure 4.9, where the geographic location of the stations can be seen in a Spain physical map [5], we conclude for each cluster:

1. Cold climate (purple): This cluster includes the alpine climate and the cold continental climate is easily distinguished because of the lower overall mean temperature. We can assign to this cluster the alpine climate, as most of the locations are near mountain ridges.
2. Mediterranean climate (orange): We can see that the locations of these stations spawn across the mediterranean coastline and the Balearic Islands. In the dispersion diagram, they have low both first and second principal component scores, indicating a higher mean temperature and accentuated differences between summer and winter.
3. Oceanic climate (green): the stations are located near the Atlantic Ocean on the north, and its scores indicate a high maritime influence.
4. Subtropical climate (red): strongly suggested because of extremely low first principal component scores that indicate hot weather, together with a very strong maritime influence and the fact that the stations in this group are all located in the Canary Islands.
5. Warm continental climate (yellow): indicated by the lower overall mean and low maritime influence. Moreover, the stations are located inland.

4.3. Phonemes dataset

The Phonemes dataset contains 4509 data of five different phonemes: 872 data of the phoneme “sh” as in “she”, 757 of “dcl” as in “dark”, 1163 of “iy” as the vowel in “she”, 695 of “aa” as the vowel in “dark”, and 1022 of “ao” as the first vowel in “water”. Originally, each datum is a recording of 32ms with a sampling rate of 16kHz. The data we are treating, however, is the log-periodogram [19] of the

(a) Spain, mainland

(b) Spain, Canary Islands

Figure 4.9: Stations mapped to the Spain map and divided in 5 clusters.

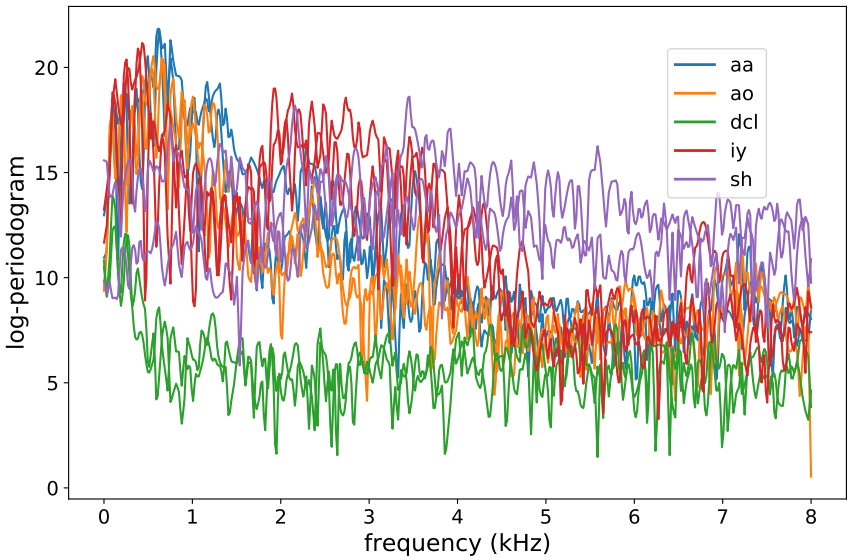
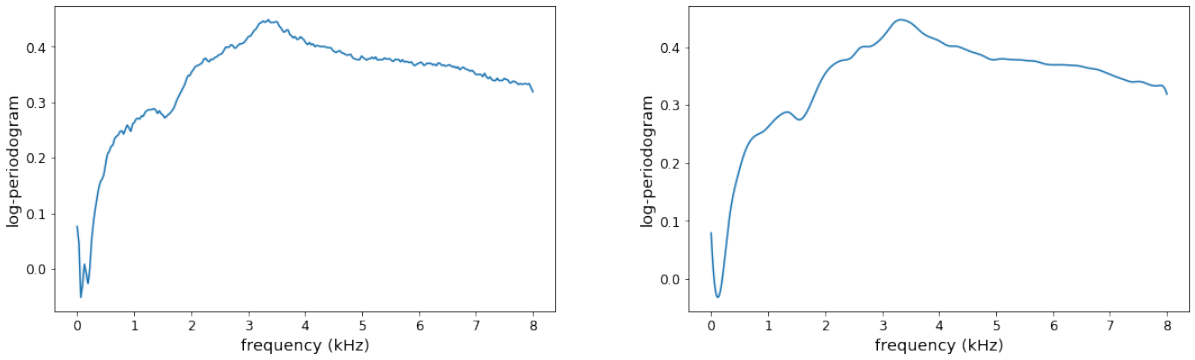


Figure 4.10: Ten curves of the Phonemes dataset, two of each phoneme.

original data. The log-periodogram is an estimation of the spectral density of each datum over the frequency spectrum. That is, the intensity of the phoneme with respect to frequency. The domain of the log-periodograms is truncated at 8kHz, starting at 0kHz, with 256 sample points equally spaced over this domain. In Figure 4.10 ten curves are shown, two of each phoneme. We can see that the curves present constant irregularities. This is the reason why this dataset is chosen to illustrate the smoothing effect of the principal components in FPCA with regularization, as described in section 2.7. For a detailed analysis of this dataset using trimmed mean and depth based median as explained in Section 2.8, please view Appendix A.

First, We apply regularization to the first principal component in both representations, for different regularization parameters, and view how the principal component curve varies as the parameter increases. The differences between the smoothed components and the original unsmoothed component are also shown. Then, we compare the smoothed principal component curves obtained in three different ways:

1. We represent the dataset in a basis representation, then we smooth the entire dataset, and finally, we apply FPCA to the smoothed dataset.
2. We represent the dataset in a basis representation, then we apply FPCA with regularization to the dataset.
3. We apply FPCA with regularization to the dataset in discrete representation.



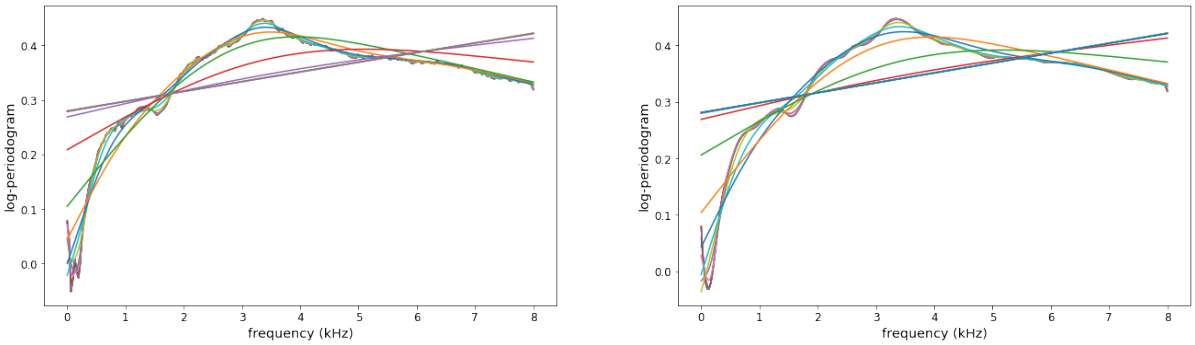
(a) Discrete representation

(b) Basis representation

Figure 4.11: First principal component of the Phonemes dataset, in both basis and discrete representations.

4.3.1. Regularized first principal component

In Figure 4.11 we show the first principal component for both representations. In this case, the chosen basis is BSplines with 55 basis functions. As expected, there is a large amount of irregularity present in the first principal component in both representations.



(a) Discrete representation

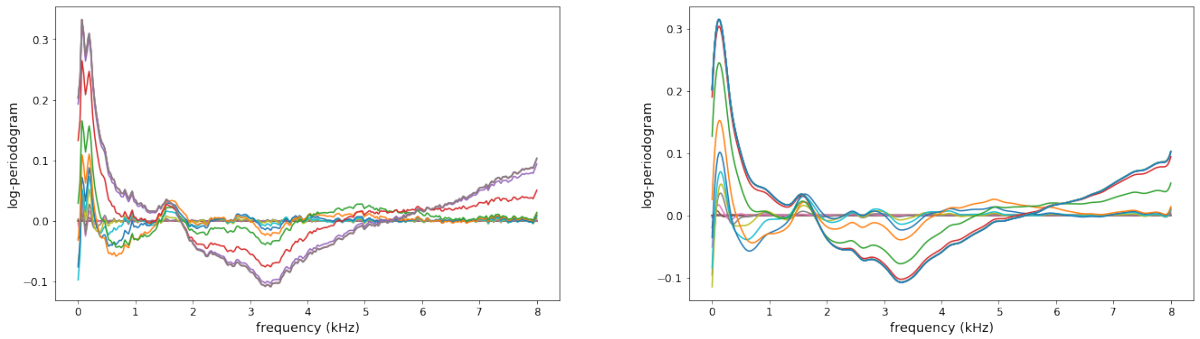
(b) Basis representation

Figure 4.12: First principal component of the Phonemes dataset, regularized for different regularization parameters.

The penalization to the curvature described in Equation 2.23 is applied to the data for both discrete and basis representations. The first principal component curve is computed for different regularization values, on a logarithmic scale. We will visualize its effect over the amount of smoothing applied showing the differences of the smoothed curves with the original curve. In Figure 4.12 the regularized first principal components are shown. In the discrete case, the regularization parameter (λ) varies from 10^{-5} to 10^{11}, and in the basis case from 10^{-10} to 10^{9}.

We can observe that as the parameter increases, the smoothing effect is more notable. When the

parameter tends to infinity, the curve becomes a straight line, as explained in Section 2.7. This is because the curvature is penalized.



(a) Discrete representation

(b) Basis representation

Figure 4.13: Differences between the regularized first principal component, and the original principal component, in both representations.

In Figure 4.13 we can see the differences with the original principal component of the regularized principal components. An increment in the regularization parameter results in an increment in the difference. It stops increasing when the parameter is so big that the curves are straight lines.

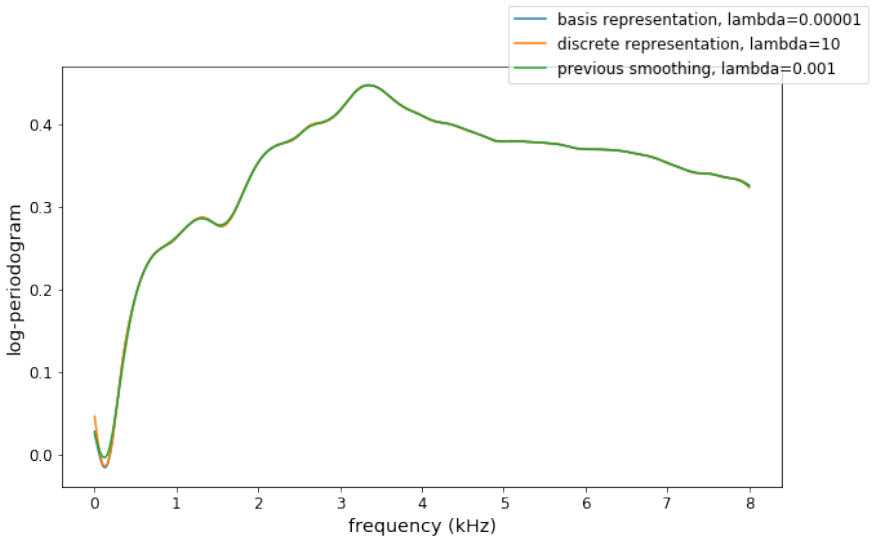


Figure 4.14: Comparison of regularized curve in three different ways.

The conclusion is that the regularization produces similar principal component curves independently of the representation or when the smoothing is applied, either during the FPCA process or as a preprocessing step. This is shown in Figure 4.14.

If we use an adequate basis this is theoretically expected. Because the data curves in the basis representation should be similar to the original data curves. As we are theoretically penalizing the curvature of the underlying function, the effect should be the same regardless of the representation.

5

Conclusions and future work

During this academic year, principal component analysis for functional data has been added to the library `scikit-fda` [8]. This is a rather important technique for FDA because it projects the functional data, originally in the infinite-dimensional space L^2, into a finite subspace. The theoretical aspects of FPCA are explored in Chapter 2. Then, analysis of several datasets has been performed using FPCA in Chapter 3, which shows how reducing the number of data attributes helps us understand them better. Other features like trimmed mean and depth-based median are also implemented.

Amongst the tools that allowed the development of this work, the most important one is Github. Integrated into Github is the continuous integration tool, Travis CI, which was the key to eliminate bugs and ensure code robustness. Team communication was also important during the year, and it was achieved by the weekly meetings and within the code review process. Discussing the technical difficulties and design decisions with other team members was particularly constructive towards the project.

For future work regarding the library, other dimensionality reduction methods may be considered. For example, partial least squares (PLS) for functional data might be the next step. Another possibility would be PCA for multivariate functional data, however, it requires other groundwork in the library to support multivariate functional data.

During the development of this project, I have acquired or improved several skills. One of them is the ability to cooperate within an open-source project. Furthermore, I learned in-depth about the capabilities of Github, such as continuous integration using Travis CI. On the other hand, I also learned concepts regarding functional data analysis, and how this tool can help us understand the data.

To conclude, this project has served as the consolidation of this degree. Most skills acquired during the degree were put to use in a practical project, such as coding practices, object-oriented programming, data structure, module design, etc. Furthermore, new knowledge in functional data analysis was also acquired.

Bibliography

[1] M. Carbajo Berrocal. FDA-PY: desarrollo de un paquete Python para el análisis de datos funcionales. *Universidad Autónoma de Madrid*, 2018.

[2] J.B. Conway. *A Course in Functional Analysis*. Graduate Texts in Mathematics. Springer New York, 1994. URL: https://books.google.es/books?id=ix4P1e6AkeIC.

[3] Departamento de Producción de la Agencia Estatal de Meteorología de España y el Departamento de Meteorologia e Clima del Instituto de Meteorologia de Portugal. *Atlas Climático Ibérico*. 2011.

[4] V. Driessen. A successful git branching model. https://nvie.com/posts/a-successful-git-branching-model/. [Online; accessed 08-february-2020].

[5] Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and GIS User Community. World Imagery. http://server.arcgisonline.com/arcgis/rest/services/World_Imagery/MapServer. [Online; accessed 03-may-2020].

[6] M. Febrero-Bande and M. Oviedo de la Fuente. Statistical computing in functional data analysis: The R package fda.usc. *Journal of Statistical Software*, 51(4):1–28, 2012. URL: http://www.jstatsoft.org/v51/i04/.

[7] R. Fraiman and G. Muniz. Trimmed means for functional data. *TEST: An Official Journal of the Spanish Society of Statistics and Operations Research*, 10(2):419–440, December 2001. URL: https://ideas.repec.org/a/spr/testjl/v10y2001i2p419-440.html, doi:10.1007/BF02595706.

[8] GAA-UAM. scikit-fda. *GitHub repository*, 2020. doi:10.5281/zenodo.3468127.

[9] A. Hernando Bernabé. Development of a Python package for Functional Data Analysis. Depth measures, applications and clustering. *Universidad Autónoma de Madrid*, 2019.

[10] D.W. Hosmer, S. Lemeshow, and R.X. Sturdivant. *Applied Logistic Regression*. Wiley Series in Probability and Statistics. Wiley, 2013. URL: https://books.google.es/books?id=64JYAwAAQBAJ.

[11] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL: http://www.scipy.org/.

[12] N. Krämer, A.-L. Boulesteix, and G. Tutz. Penalized partial least squares with applications to b-spline transformations and functional data. *Chemometrics and Intelligent Laboratory Systems*, 94:60–69, 11 2008. doi:10.1016/j.chemolab.2008.06.009.

[13] Instituto Geográfico Nacional. Tipos de clima. https://www.ign.es/espmap/mapas_clima_bach/Mapa_clima_13.htm. [Online; accessed 03-may-2020].

[14] T. Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. [Online; accessed 5/3/2020]. URL: http://www.numpy.org/.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[16] C. Ramos Carreño. scikit-fda: A Python package for Functional Data Analysis. *III International Workshop on Advances in Functional Data Analysis*, 2019.

[17] J. Ramsay and B.W. Silverman. *Functional Data Analysis*. Springer Series in Statistics. Springer, 2005. URL: https://books.google.es/books?id=mU3dop5wY_4C.

[18] J. O. Ramsay, Hadley Wickham, Spencer Graves, and Giles Hooker. *fda: Functional Data Analysis*, 2018. R package version 2.4.8. URL: https://CRAN.R-project.org/package=fda.

[19] A. Schuster. On the investigation of hidden periodicities with application to a supposed 26 day period of meteorological phenomena. *Terrestrial Magnetism*, 3(1):13–41, 1898. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/TM003i001p00013, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/TM003i001p00013, doi:10.1029/TM003i001p00013.

[20] J. Shlens. A tutorial on principal component analysis. *CoRR*, abs/1404.1100, 2014. URL: http://arxiv.org/abs/1404.1100, arXiv:1404.1100.

[21] R. D. Tuddenham and M. M. Snyder. Physical growth of california boys and girls from birth to eighteen years. *Publications in child development. University of California, Berkeley*, 1(2):183—364, 1954. URL: http://europepmc.org/abstract/MED/13217130.

Appendices

A

Trimmed mean and depth based median for the dataset Phonemes

In this appendix a detailed analysis of the Phonemes dataset, which was used to illustrate the regularization effects over the principal components in the Section 4.3, is shown.

A.1. Overall trimmed mean for the dataset Phonemes

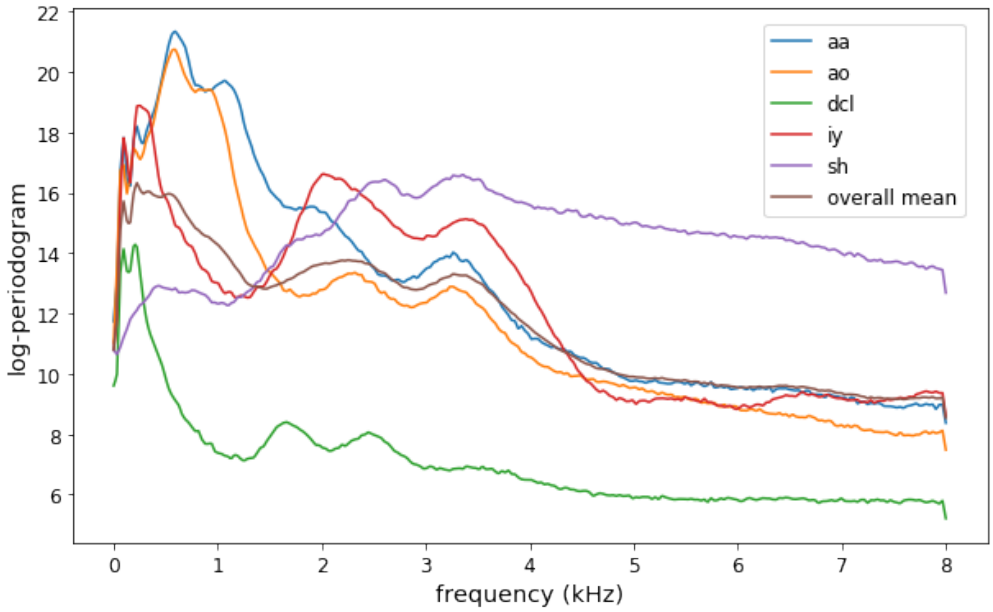


Figure A.1: The overall mean and individual mean of each phoneme.

The mean curves of each phoneme, as well as the overall mean of the entire dataset is visualized in Figure A.1. The mean curves of the vowels (“aa”, “ao” and “iy”) are similar while the mean curves of “dcl” and “sh” differ. The phoneme “dcl” is clearly lower in intensity than the other phonemes, and the phoneme “sh” has a much higher intensity than the rest from 3kHz to 8kHz.

The first and foremost way to apply trimmed mean is to the entire dataset. The trimmed mean curves for different trim percentages are shown in Figure A.2. From 0 to 4kHz, the more we trim, the more the mean curve increases. From 4 to 8 kHz, we observe a rather interesting effect, as the mean curve first

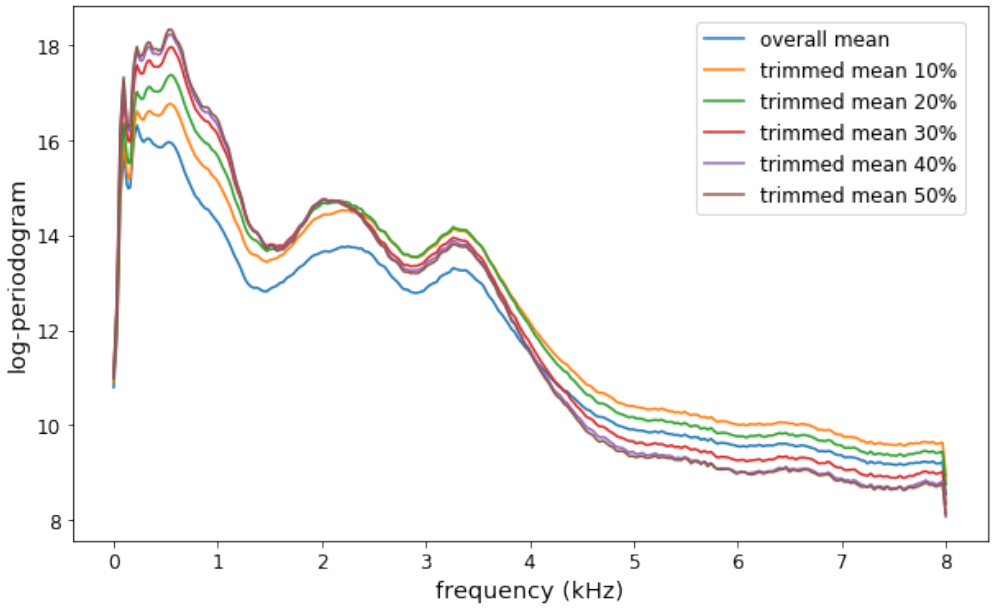


Figure A.2: Trimmed means for the Phonemes dataset, with different trim percentages.

increases and then decreases. This behavior is caused by the similarities that the vowels aa, ao share. As the three of them composes more than the half of the data sample, the data is more centered around these phonemes. As a result, they tend to have higher depth values, and are barely eliminated. When we trim, data of the phonemes “dcl” and “sh” are eliminated first, resulting in the behavior described above.

Because both phonemes have a lower intensity in the first half of the domain, the more we eliminate, the more the mean curve increases. For example, when we trim $20\,\%$ of the data, the number of dcl samples decreases from 757 to 146 and the number of sh samples decreases from 872 to 586, well above $20\,\%$. The other phonemes are barely dropped. Also, the phoneme “dcl” is even less centered than the phoneme “sh”. This explains the rise and fall of the second half of the domain, because when we trim with a lesser percentage, the data pertaining to the group “dcl” are trimmed first, which makes the overall mean increase. However, once we run out of “dcl” samples to trim, we start eliminating “sh” curves that have a higher intensity in the second half of the domain, resulting in the drop.

A.2. The trimmed mean for each phoneme

The overall trimmed mean is functioning as expected. However, it is not appropriate for this dataset because we do not want to drop one specific phoneme group. The objective here is to use the trimmed mean to eliminate outliers from the overall mean, and clearly not all “dcl” data are outliers of its group. Therefore, it is more sensible to use trimmed mean on each group and then gather the results.

In figure A.3 the individual trimmed mean for each phoneme is plotted, using a trim percentage of

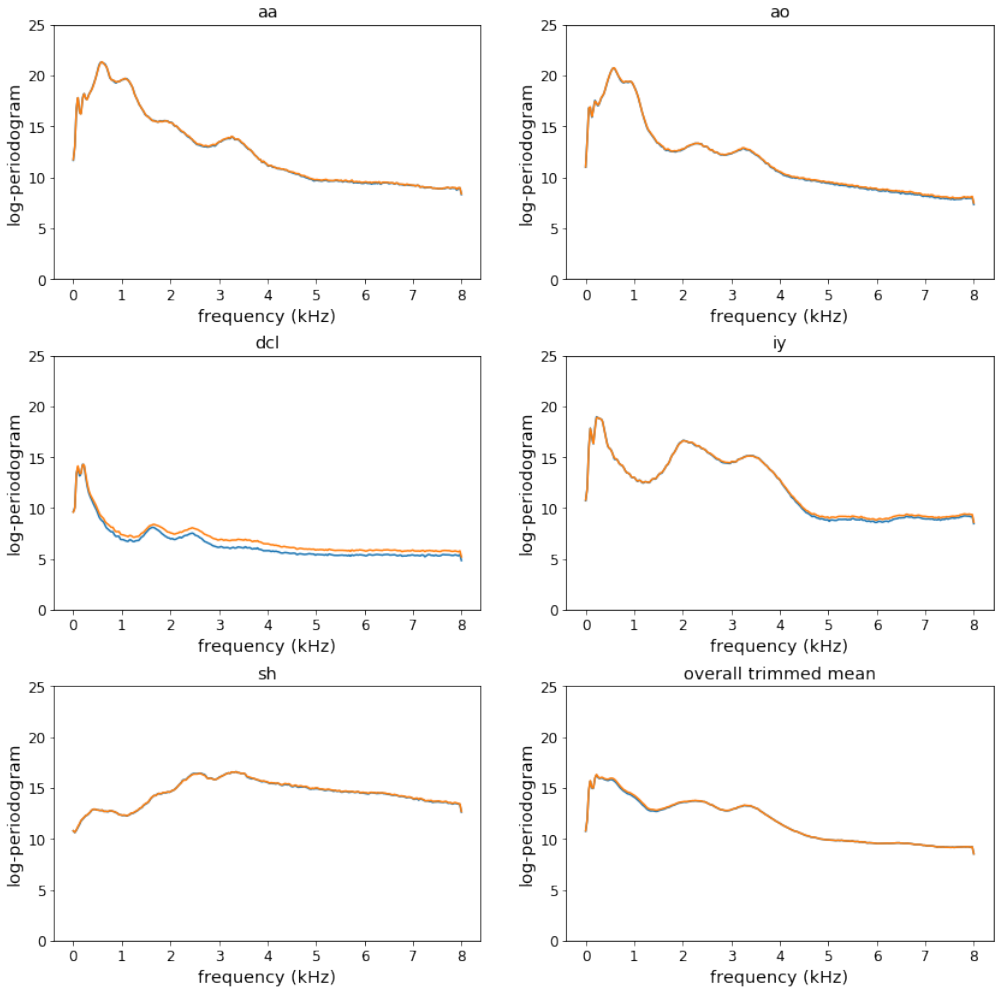
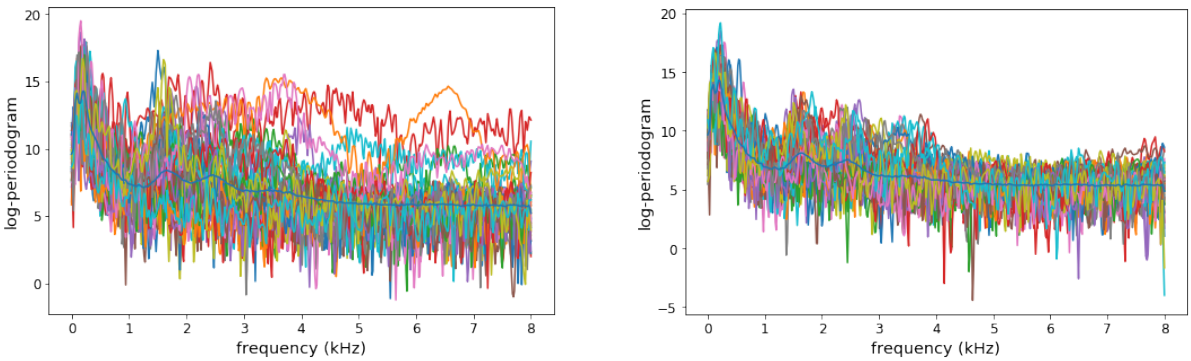


Figure A.3: The trimmed mean functions for each phoneme compared with original mean functions.



(a) Before trim

(b) After trim

Figure A.4: 50 random dcl data curves before and after trimming

$20\,\%$. The phoneme “dcl” stands out as it has more difference with the original mean. This is because it has more outliers than other phonemes, which can be seen in Figures A.4(a) and A.4(b). The trimmed mean is generally lower because the outliers have a higher intensity.

A.2.1. Depth-based median for the Phonemes dataset

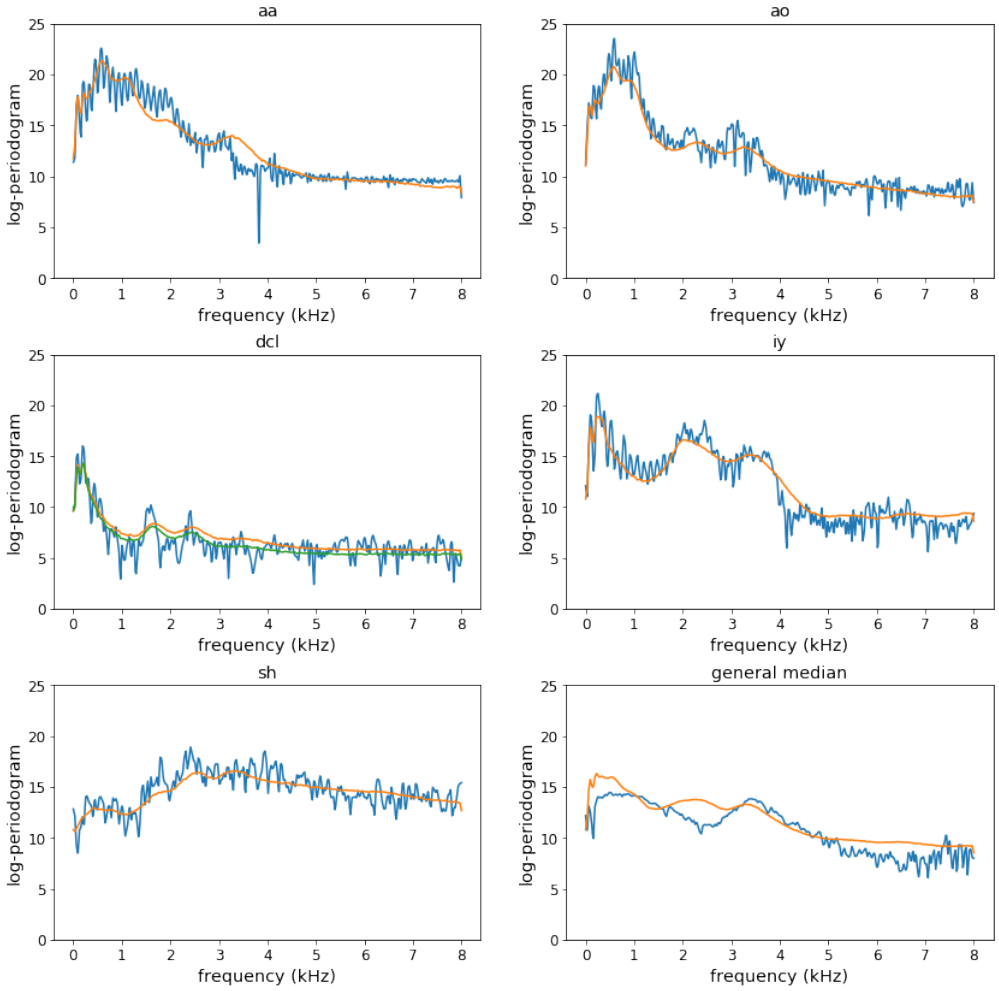


Figure A.5: Comparison of median curves with mean curves for the Phonemes dataset.

We present the depth-based medians for each phoneme, with each phoneme's mean function in Figure A.5. For the dcl phoneme the trimmed mean function is also plotted, as it is the only phoneme with visible difference.

The main conclusion to be drawn here is that the mean curve is always close to the median curve, indicating that the mean functions are good representations of the characteristics of the sample data. The only exception is again the phoneme dcl, as it is closer to the trimmed mean.

B

Simulation result notebook

In this appendix we show the notebook that was developed to obtain the figures in Chapter 4 and Appendix A. From page 1 to 10 we treat the Berkeley Growth Study. From page 11 to 19 we analyze the AEMET weather dataset. And from page 19 to 38 the analysis of the Phonemes dataset is shown.

Simulation result graphs

June 4, 2020

[1]:
```
import numpy as np
import skfda
from skfda.preprocessing.dim_reduction.projection import FPCA#,␣
 ↪RegularizationParameterSearch, FPCARegularizationCVScorer
from skfda.representation import FDataBasis, FDataGrid
from skfda.datasets._real_datasets import *
from matplotlib import pyplot as plt
import matplotlib as mpl
from skfda.representation.basis import Fourier, BSpline, Monomial
from sklearn.decomposition import PCA

import skfda.preprocessing.smoothing.kernel_smoothers as ks
import skfda.preprocessing.smoothing.validation as val

from skfda.exploratory.visualization.clustering import (
    plot_clusters, plot_cluster_lines, plot_cluster_bars)
from skfda.ml.clustering import KMeans, FuzzyCMeans

from matplotlib.lines import Line2D
```

[2]:
```
def plot_axes_labels(es=False, axes_labels=None, axes_labels_es=None):
    if not es and axes_labels is not None:
        plt.xlabel(axes_labels[0])
        plt.ylabel(axes_labels[1])
    if es and axes_labels_es is not None:
        plt.xlabel(axes_labels_es[0])
        plt.ylabel(axes_labels_es[1])
```

[16]:
```
def plot_components(components,
                    axes_labels=None,
                    axes_labels_es=None,
                    n_components=2,
                    title=None,
                    es=False,
                    bbox_to_anchor=None,
                    save=None):
    fig = plt.figure(figsize=(8,5))
```

```
    manual_labels = ['curva de la primera componente', 'curva de la segunda␣
↪componente']
    for i in range(n_components):
        component_label = manual_labels[i] if es else 'Principal component␣
↪curve ' + str(i+1)
        components[i].plot(fig, label=component_label)

    if bbox_to_anchor is not None:
        fig.legend(loc='center left', bbox_to_anchor=bbox_to_anchor)
    else:
        fig.legend(loc='lower right')

    plot_axes_labels(es, axes_labels, axes_labels_es)


    if title is None:
        if isinstance(components, FDataBasis):
            title_ = 'Representación en bases' if es else None
        else:
            title_ = 'Discretización' if es else None
    else:
        title_ = title
    plt.suptitle(None)
    plt.title(title_)
    if save is not None:
        plt.savefig(save + ".pdf")
    plt.show()
```

[18]:
```
def plot_perturbations_over_mean(fd, components, index,  multiple,
                                 axes_labels=None,
                                 axes_labels_es=None,
                                 es=False, bbox_to_anchor=None, save=None):
    fig = plt.figure()
    mean_fd = fd.mean()
    mean_fd = mean_fd.concatenate(mean_fd[0] + multiple * components[index])
    mean_fd = mean_fd.concatenate(mean_fd[0] - multiple * components[index])
    mean_fd[0].plot(fig=fig,label='media' if es else 'mean')
    mean_fd[1].plot(fig=fig,label='perturbación positiva' if es else 'positive␣
↪perturbation')
    mean_fd[2].plot(fig=fig,label='perturbación negativa' if es else 'negative␣
↪perturbation')

    plot_axes_labels(es, axes_labels, axes_labels_es)
    manual_title = ['Primera componente principal', 'Segunda componente␣
↪principal']
    plt.title(manual_title[index] if es else None)
    plt.suptitle(None)
```

```
    if bbox_to_anchor is not None:
        plt.legend(bbox_to_anchor=bbox_to_anchor)
    else:
        plt.legend()
    if save is not None:
        plt.savefig(save + ".pdf")
    plt.show()
```

[34]:
```
def plot_dispersion_diagram(scores, target, names, axes=[0,1], axes_labels=None,
                            axes_labels_es=None, title=None, es=False,␣
↪colors=['C0', 'C1'],
                            save=None):
    n_classes = len(set(target))
    first_score = scores[:, axes[0]]
    second_score = scores[:, axes[1]]
    for i in range(n_classes):
        plt.scatter(first_score[target==i],second_score[target==i],␣
↪label=names[i], color=colors[i])
    plt.legend()
    plot_axes_labels(es, axes_labels, axes_labels_es)
    if save is not None:
        plt.savefig(save + ".pdf")
    plt.show()
```

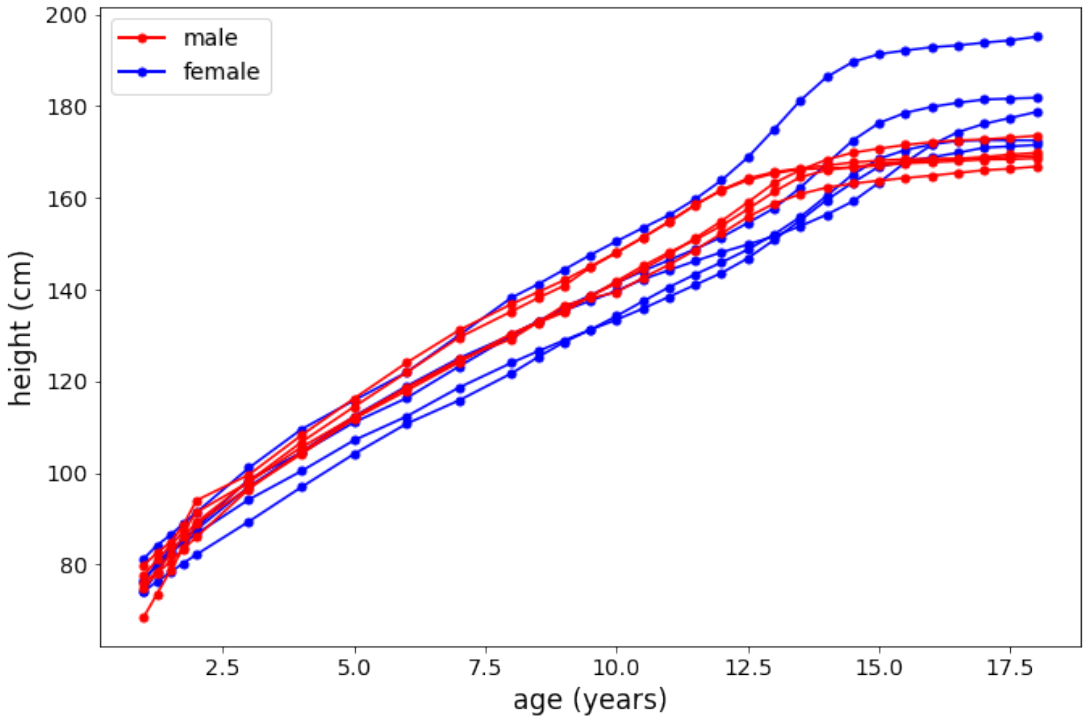
[24]:
```
growth_fd = fetch_growth()['data']
growth_fd = growth_fd[:5].concatenate(growth_fd[-5:])
```

[7]:
```
mpl.rcParams['axes.labelsize'] = 'large'
mpl.rcParams['font.size'] = 14.0
```

[8]:
```
growth_axes_labels_es = ['Edad (años)', 'Altura (cm)']
growth_axes_labels = ['Age (years)', 'Height (cm)']
es = False
```

[9]:
```
mpl.rcParams['figure.figsize'] = np.array([12,8])/1.1
linestyle = mpl.rcParams['lines.linestyle']
marker = mpl.rcParams['lines.marker']
markersize = mpl.rcParams['lines.markersize']
mpl.rcParams['lines.linestyle'] = '-'
mpl.rcParams['lines.marker'] = 'o'
mpl.rcParams['lines.markersize'] = 5

#growth_fd.plot(linestyle='--', marker='o')
plt.plot(growth_fd.sample_points[0], np.transpose(np.squeeze(growth_fd[:5].
↪data_matrix)), color='b', label='male')
```

```
plt.plot(growth_fd.sample_points[0], np.transpose(np.squeeze(growth_fd[5:].
 ↪data_matrix)), color='r', label='female')
plt.title(None)
custom_lines = [Line2D([0], [0], color='r', lw=2),
                Line2D([0], [0], color='b', lw=2)]
plt.legend(custom_lines, ['male', 'female'])
plt.xlabel('age (years)')
plt.ylabel('height (cm)')
plt.savefig("berkeley-growth.pdf")
plt.show()
```



[10]:
```
mpl.rcParams['lines.linestyle'] = linestyle
mpl.rcParams['lines.marker'] = marker
mpl.rcParams['lines.markersize'] = markersize
mpl.rcParams['axes.labelsize'] = 'large'
mpl.rcParams['figure.figsize'] = [8,5]
mpl.rcParams['font.size'] = 14.0
```

[11]:
```
berkeley = fetch_growth()
fd = berkeley['data']
fd.axes_labels = growth_axes_labels_es if es else growth_axes_labels
y = berkeley['target']
```

[12]:
```
n_components=5
axes_labels = fd.axes_labels
```

[13]:
```
fd_basis = fd.to_basis(BSpline(n_basis=8))
fpca_basis = FPCA(n_components=n_components)
fpca_basis.fit(fd_basis)
```

[13]:
```
FPCA(centering=True, components_basis=None, n_components=5, regularization=None,
     weights=None)
```

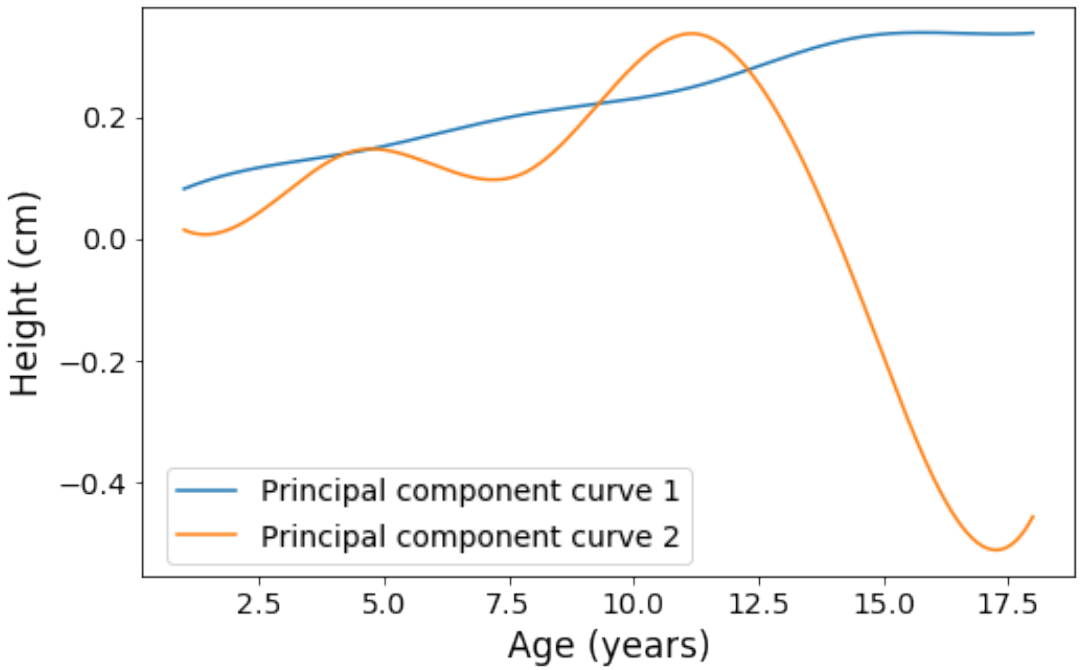
[14]:
```
fpca_basis.explained_variance_
```

[14]:
```
array([6.05542292, 0.98972476, 0.22578554, 0.08484623, 0.03445689])
```

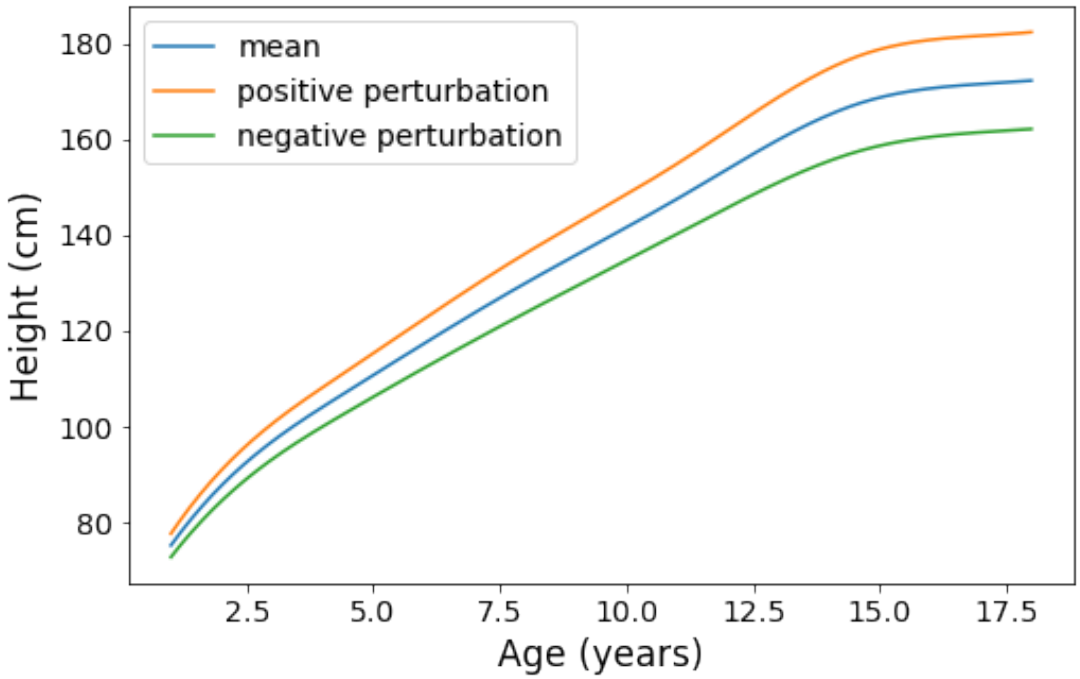
[15]:
```
fpca_basis.explained_variance_ratio_
```

[15]:
```
array([0.81623123, 0.1334084 , 0.03043441, 0.01143671, 0.00464456])
```

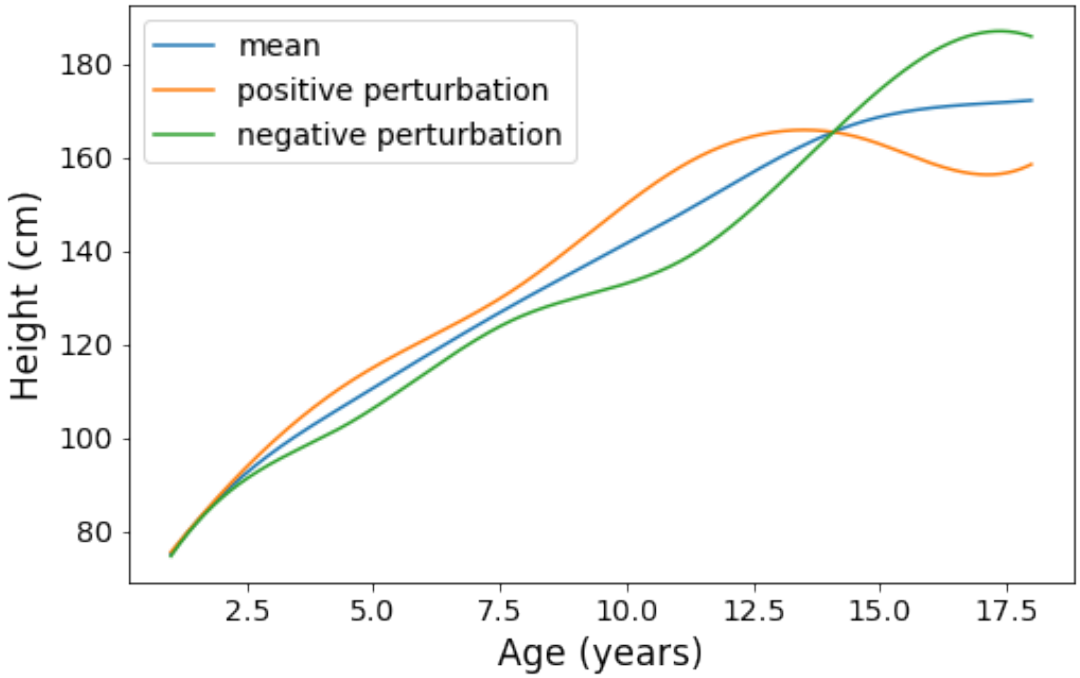
[27]:
```
plot_components(fpca_basis.components_,
                growth_axes_labels,
                growth_axes_labels_es,
                bbox_to_anchor=(0.13,0.22),
               n_components=2,
               save='berkeley_principal_components_basis')
```



[20]:
```
plot_perturbations_over_mean(fd.to_basis(BSpline(n_basis=8)), fpca_basis.
↪components_, 0, 30,
                             axes_labels=growth_axes_labels,␣
↪axes_labels_es=growth_axes_labels_es,
                             save='berkeley_variations_first_component')
```

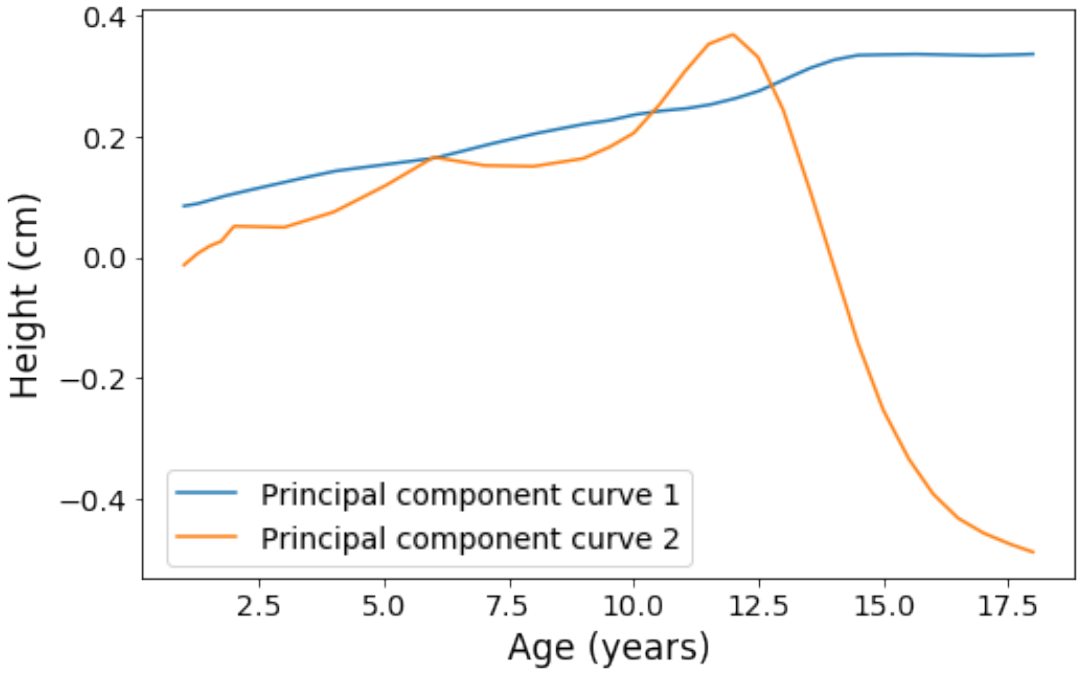


[21]:
```
plot_perturbations_over_mean(fd.to_basis(BSpline(n_basis=8)), fpca_basis.
↪components_, 1, 30,
                             axes_labels=growth_axes_labels,␣
↪axes_labels_es=growth_axes_labels_es,
                             save='berkeley_variations_second_component')
```



[22]:
```
fd = fetch_growth()['data']
fpca_grid = FPCA(n_components=2)
fpca_grid = fpca_grid.fit(fd)
```

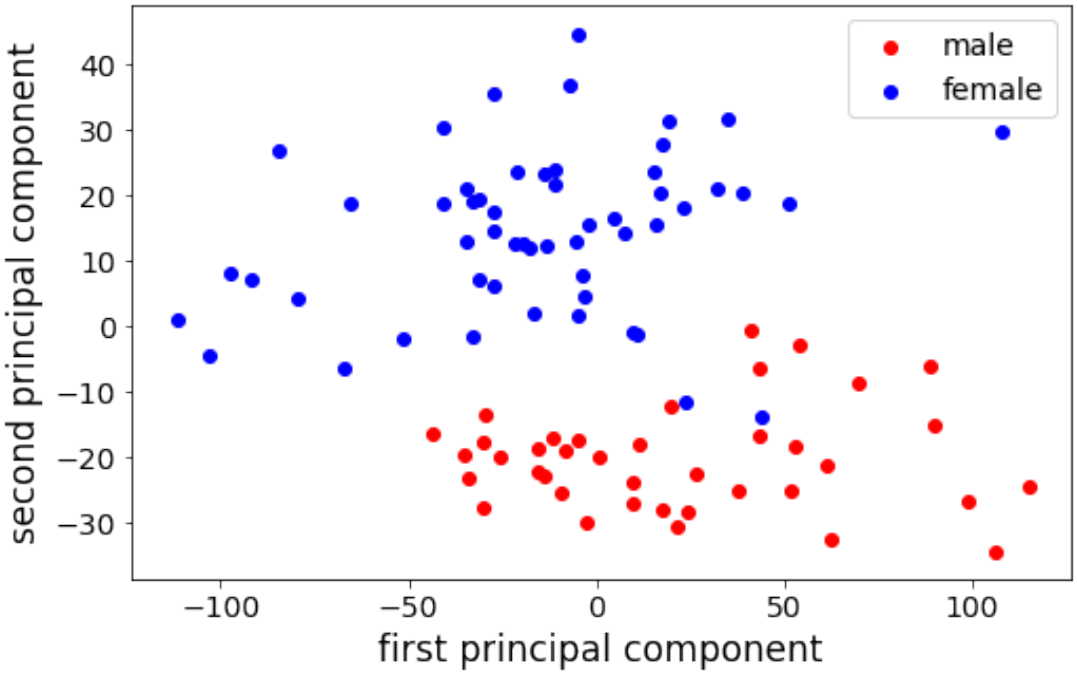
[26]:
```
plot_components(fpca_grid.components_, axes_labels=growth_axes_labels,
                bbox_to_anchor=(0.13,0.22), n_components=2,␣
 ↪save='berkeley_principal_components_grid')
```



[28]:
```
result = fpca_grid.transform(fd)
```

[29]:
```
dispersion_labels_es = ['primera componente principal', 'segunda componente␣
↪principal']
dispersion_labels = ['first principal component', 'second principal component']
```

[35]:
```
plot_dispersion_diagram(result, berkeley['target'], ['male', 'female'], [0,1],
                        axes_labels=dispersion_labels,␣
↪axes_labels_es=dispersion_labels_es, es=False,
                        colors=['r', 'b'], save='berkeley_dispersion_diagram')
```



[36]:
```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
```

[37]:
```
X = result[:, :2]
y = berkeley['target']
```

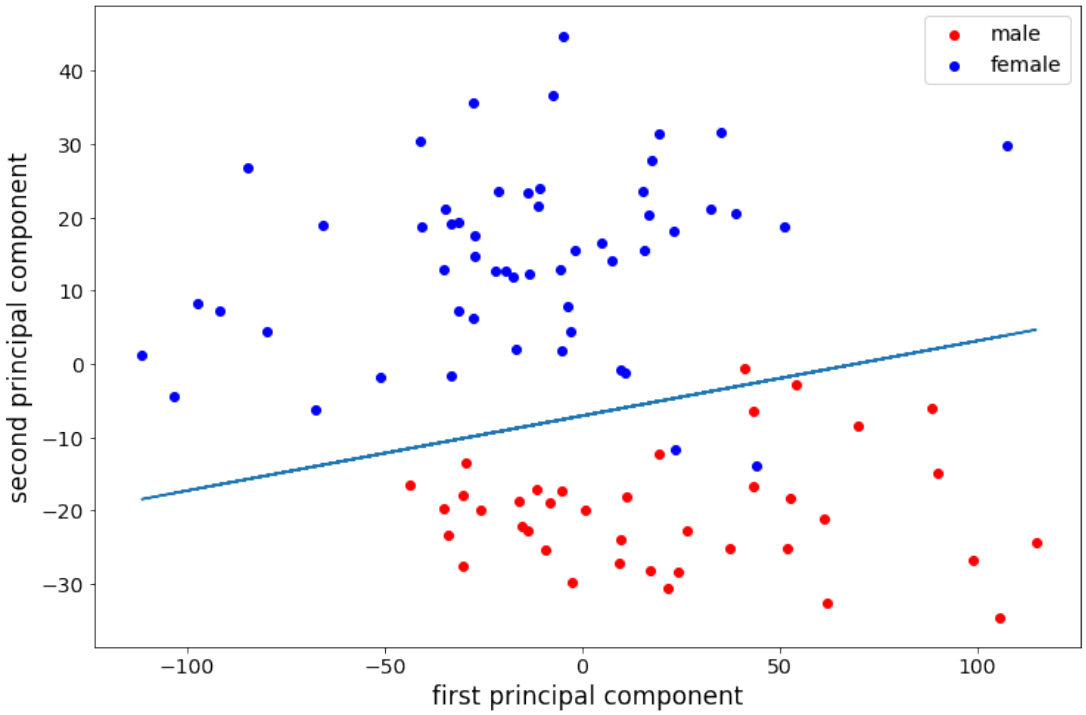
[38]:
```
logisticRegr = LogisticRegression()
logisticRegr = logisticRegr.fit(X, y)
logisticRegr.predict(X)
```

[38]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
       1, 1, 1, 1, 1])
```

[39]:
```
boundary = (-logisticRegr.intercept_[0] -
            (logisticRegr.coef_[0][0] * X[:, 0])) / logisticRegr.coef_[0][1]
```

[41]:
```
scores = result
target = y
names_es=['niños', 'niñas']
names=['male', 'female']
axes=[0,1]
```

```
axes_labels = ['first principal component',
               'second principal component']
axes_labels_es=['primera componente principal',
              'segunda componente principal']
colors = ['r', 'b']
n_classes = len(set(target))
first_score = scores[:, axes[0]]
second_score = scores[:, axes[1]]
mpl.rcParams['figure.figsize'] = [12,8]
for i in range(n_classes):
    plt.scatter(first_score[target==i],second_score[target==i], label=names[i],␣
 ↪color=colors[i])
plt.legend()
if axes_labels is not None:
    plt.xlabel(axes_labels[0])
    plt.ylabel(axes_labels[1])
plt.plot(X[:, 0], boundary)
plt.savefig('logisticRegre_berkeley.pdf')
plt.show()
mpl.rcParams['figure.figsize'] = [8,5]
```



[70]:
```
logisticRegr.score(X, y)
```

[70]:
```
0.967741935483871
```

1 Weather PCA

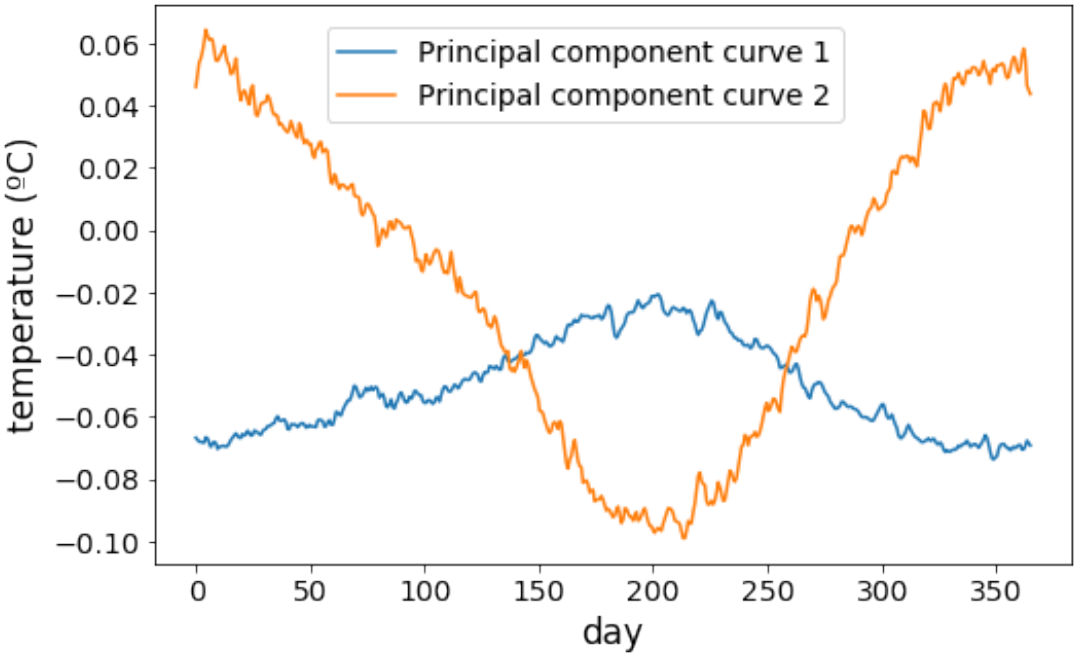
[42]:
```
aemet = fetch_aemet()
fd = aemet['data']
fd_temp = fd.coordinates[0]
y = aemet['meta']
aemet_axes_labels = ['day', 'temperature (ºC)']
aemet_axes_labels_es = ['Día', 'Temperatura (ºC)']
```

```
      ind                              name    province  altitude  year.ini  \
0    1387                          A CORUÑA    A CORUÑA        58      1980
1    1387                A CORUÑA/ALVEDRO    A CORUÑA        98      1980
2    1428  SANTIAGO DE COMPOSTELA/LABACOLLA    A CORUÑA       370      1980
3   90910                VITORIA/FORONDA       ALAVA       513      1980
4    8175             ALBACETE/LOS LLANOS    ALBACETE       704      1980
..    …                                …       …       …        …
68   2539        VALLADOLID (VILLANUBLA)  VALLADOLID       846      1980
69   1082              BILBAO/AEROPUERTO     VIZCAYA        42      1980
70   2614                         ZAMORA      ZAMORA       656      1980
71   9390                         DAROCA    ZARAGOZA       779      1980
72   9434          ZARAGOZA (AEROPUERTO)    ZARAGOZA       247      1980

    year.end  longitude   latitude
0       2009  -8.419444  43.367222
1       2009  -8.372222  43.306944
2       2009  -8.410833  42.887778
3       2009  -2.733333  42.871944
4       2009  -1.863056  38.952222
..       …        …        …
68      2009  -4.850000  41.700000
69      2009  -2.905833  43.298056
70      2009  -5.733611  41.516667
71      2009  -1.410833  41.114722
72      2009  -1.008056  41.661944

[73 rows x 8 columns]
```

[43]:
```
fpca_grid = FPCA(2)
fd_fit = fd_temp.copy(data_matrix=np.copy(np.squeeze(fd_temp.data_matrix)))
fpca_grid = fpca_grid.fit(fd_fit)
```

[44]:
```
plot_components(fpca_grid.components_,
                aemet_axes_labels,
                aemet_axes_labels_es,
```

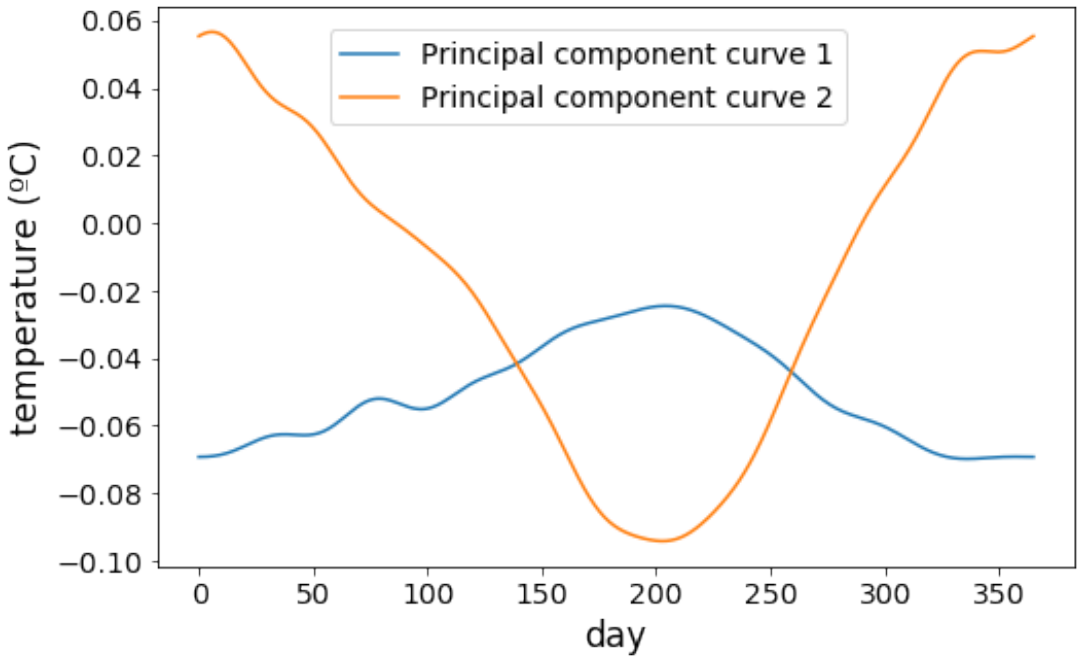
```
                bbox_to_anchor=(0.27, 0.8),
                n_components=2,
                save='aemet_principal_components_discrete')
```



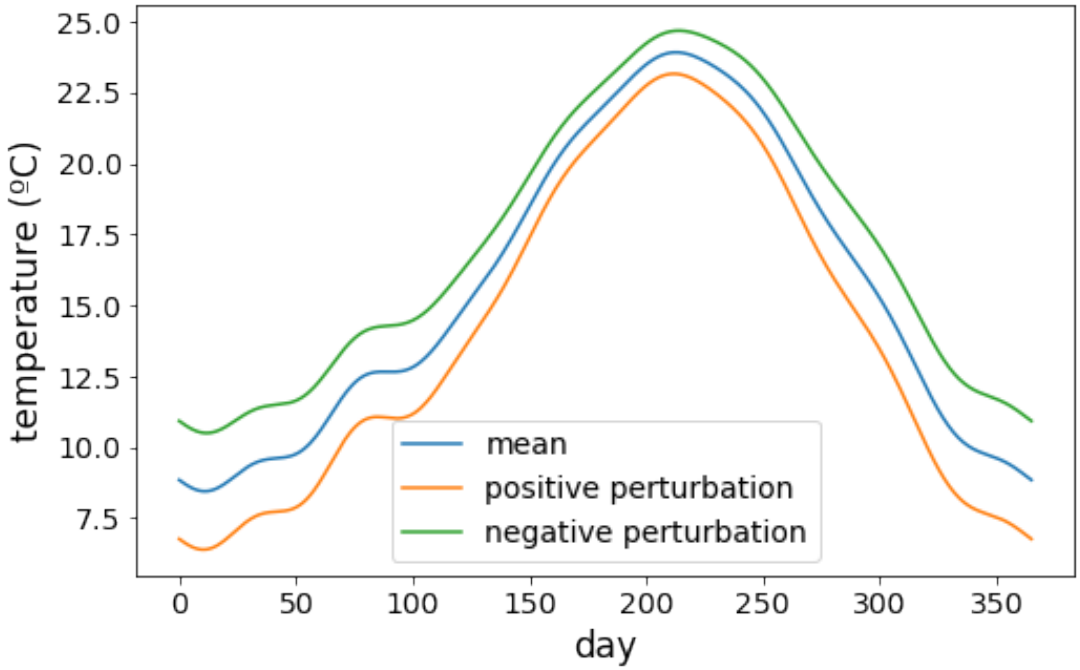
[45]:
```
temp_basis = fd_temp.to_basis(Fourier(n_basis=21))
fpca_basis = FPCA()
fpca_basis.fit(temp_basis.copy(coefficients = temp_basis.coefficients.copy()))
```

[45]:
```
FPCA(centering=True, components_basis=None, n_components=3, regularization=None,
     weights=None)
```

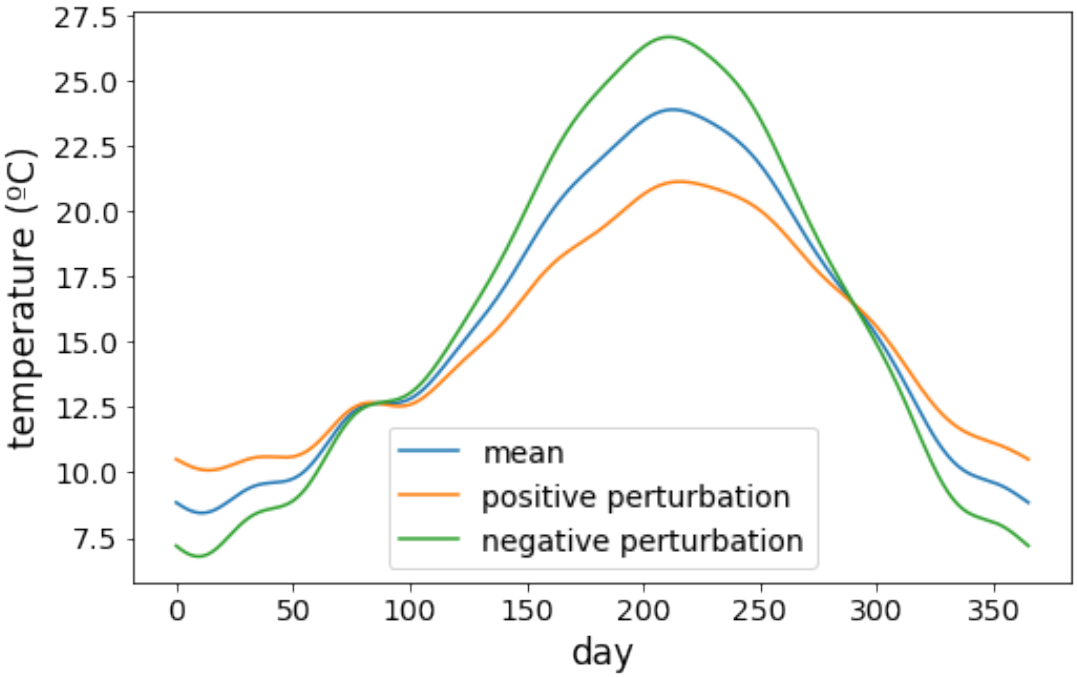
[46]:
```
plot_components(fpca_basis.components_,
                aemet_axes_labels,
                aemet_axes_labels_es,
                bbox_to_anchor=(0.27, 0.8),
                n_components=2,
                save='aemet_principal_components_basis')
```



[47]:
```
plot_perturbations_over_mean(temp_basis, fpca_basis.components_, 0, 30,
                             axes_labels=aemet_axes_labels,␣
↪axes_labels_es=aemet_axes_labels_es,
                             save='aemet_variations_first_component')
```

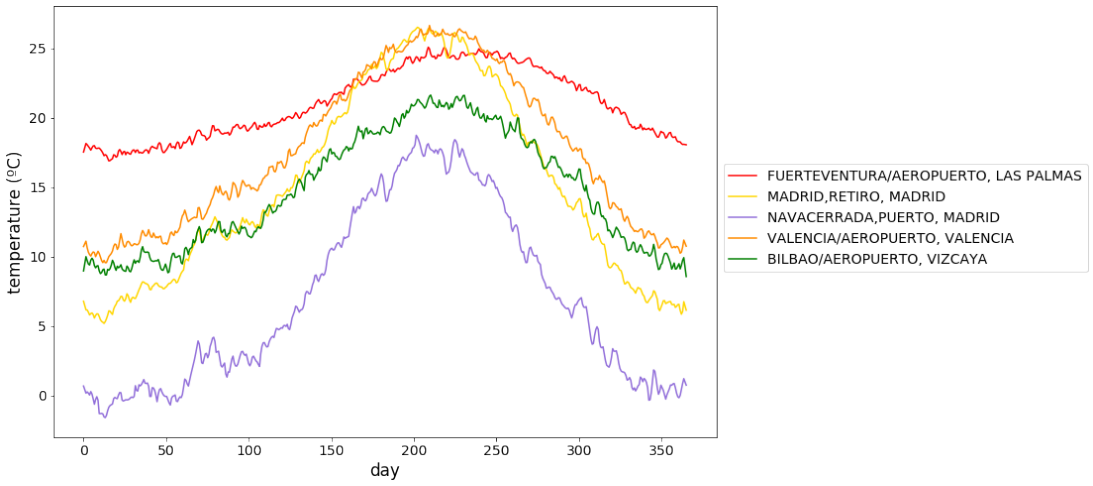


[48]:
```
plot_perturbations_over_mean(temp_basis, fpca_basis.components_, 1, 30,
                             axes_labels=aemet_axes_labels,␣
 ↪axes_labels_es=aemet_axes_labels_es,
                             save='aemet_variations_second_component')
```



[ ]:

[78]:
```
fig = plt.figure(figsize=[12,8])
plot_station_index = [33, 39, 44, 66, 69]
color_cycle=dict(zip(plot_station_index, ['red', 'gold', 'MediumPurple',␣
 ↪'DarkOrange', 'green']))
for i in plot_station_index:
    fd_temp[i].plot(fig=fig, label = aemet['meta'][i][1] + ', ' +␣
 ↪aemet['meta'][i][2], color=color_cycle[i])
plt.suptitle(None)
plt.legend(bbox_to_anchor=(1,0.65))
plt.xlabel(aemet_axes_labels[0])
plt.ylabel(aemet_axes_labels[1])
plt.savefig('aemet_five_curves.pdf')
plt.show()
```



[51]:
```
scores = fpca_grid.transform(fd_fit)
first_score = scores[:, 0]
second_score = scores[:, 1]
print(fpca_grid.explained_variance_ratio_)
print(sum(fpca_grid.explained_variance_ratio_))
```

```
[0.85539918 0.13241046]
0.9878096408352697
```

[52]:
```
def plot_dispersion_diagram(scores, target, names, axes=[0,1],␣
↪axes_labels=None, title=None):
    n_classes = len(set(target))
    first_score = np.squeeze(scores.data_matrix)[:, axes[0]]
    second_score = np.squeeze(scores.data_matrix)[:, axes[1]]
    for i in range(n_classes):
        plt.scatter(first_score[target==i],second_score[target==i],␣
↪label=names[i])
    plt.legend(loc='center left', bbox_to_anchor=(1, 1))
    if axes_labels is not None:
        plt.xlabel(axes_labels[0])
        plt.ylabel(axes_labels[1])
    plt.show()
```

[53]:
```
def get_cluster_colors(color_cycle, predictions):
    cluster_color_cycle = {}

    for i in list(color_cycle.keys()):
        cluster_color_cycle[predictions[i]] = color_cycle[i]
    return cluster_color_cycle
```

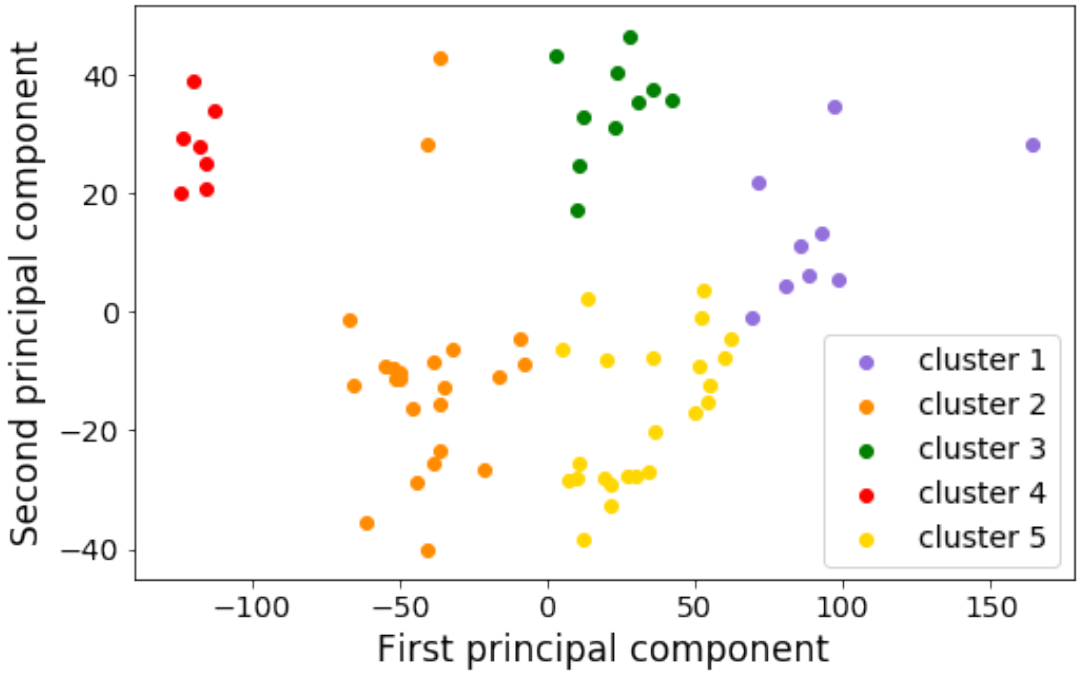
using 5 clusters, we run the algorithm 50 iterations and use the best result

[54]:
```
n_clusters=5
kmeans = skfda.ml.clustering.KMeans(n_clusters=n_clusters, n_init=50)
kmeans.fit(fd_temp)

aemet_cluster_5 = kmeans.predict(fd_temp)
print(aemet_cluster_5)
```

```
[2 2 2 0 4 1 1 1 2 2 1 1 1 1 1 4 1 0 1 1 2 1 4 1 4 4 4 4 0 2 2 4 4 3 3 3 0
 4 4 4 4 4 4 4 0 1 1 1 1 4 4 2 4 0 3 0 3 3 1 3 1 1 0 1 1 1 1 4 0 2 4 4 4]
```

[55]:
```
cluster_color_cycle = get_cluster_colors(color_cycle, aemet_cluster_5)
```

[56]:
```
for i in range(n_clusters):
    indexes = list(color_cycle.keys())
    for j in range(len(color_cycle)):
        if (aemet_cluster_5==i)[indexes[j]]:
            color = color_cycle[indexes[j]]
            break
    plt.scatter(first_score[aemet_cluster_5==i],
                second_score[aemet_cluster_5==i],
                label='cluster ' + str(i+1),
                color=color)
plt.legend(bbox_to_anchor=(1, 0.45))
plt.xlabel('First principal component')
plt.ylabel('Second principal component')
plt.savefig('aemet_dispersion_diagram_cluster.pdf')
plt.show()
```



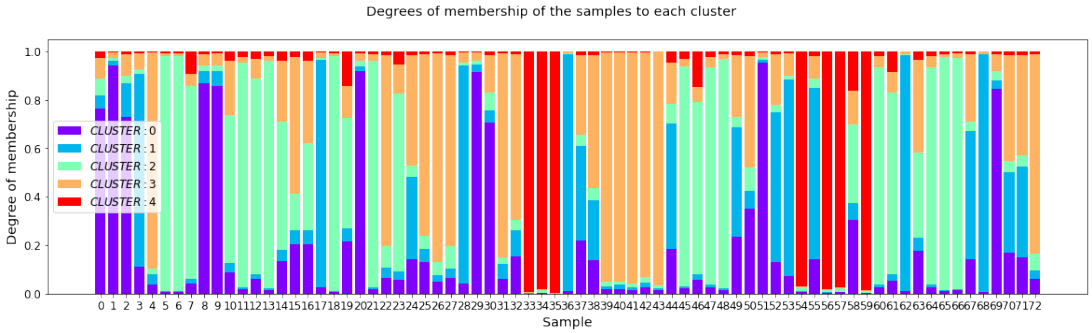
[58]:
```
longitudes = aemet['meta'][:, 4]
latitudes = aemet['meta'][:, 5]
```

[60]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
```

[346]:
```
n_clusters=5
fuzzycmeans = skfda.ml.clustering.FuzzyCMeans(n_clusters=n_clusters, n_init=50)
fuzzycmeans.fit(fd_temp)

aemet_cluster_fuzzy = fuzzycmeans.predict(fd_temp)
```

```
[0 0 0 3 4 5 5 5 0 0 5 5 5 5 5 4 4 3 5 5 0 5 4 5 1 4 4 4 3 0 0 1 1 2 2 2 3
 1 1 4 4 4 4 4 3 5 5 5 5 1 4 0 1 1 2 3 2 2 5 2 5 5 3 4 5 5 5 1 3 0 1 1 4]
```

[363]:
```
fig = plt.figure(figsize=[20,5])
plot_cluster_bars(fuzzycmeans, fd_temp, fig=fig)
plt.show()
```



[61]:
```
station_number = np.array([str(i) for i in range(fd_temp.n_samples)])
```

[63]:
```
annotations = False
fig = plt.figure(figsize=(18,18))
m = Basemap(llcrnrlon=-10,llcrnrlat=34.9800, urcrnrlon=5, urcrnrlat=44.8000,␣
 ↪projection='merc', resolution="h", epsg=5520)
#http://server.arcgisonline.com/arcgis/rest/services
m.arcgisimage(service='ESRI_Imagery_World_2D', xpixels = 1500)
m.drawcountries(linewidth=2)
x, y = m(longitudes, latitudes)
plot_station_index = [39, 44, 66, 69]
for i in range(n_clusters):
    plt.scatter(x[aemet_cluster_5==i],
                y[aemet_cluster_5==i],
                label='cluster ' + str(i+1),
                s=80,
                color=cluster_color_cycle[i],
               edgecolors = 'white')
    indices = station_number[aemet_cluster_5==i]
    for station in plot_station_index:
        plt.annotate(aemet['meta'][station][1], (x[station], y[station]),␣
 ↪color='LightGrey')
    if annotations:
        for j in range(len(x[aemet_cluster==i])):
            plt.annotate(indices[j],(x[aemet_cluster_5==i][j],␣
 ↪y[aemet_cluster_5==i][j]), color='white')

plt.legend()
plt.show()
```

```
/home/hzzhyj/.local/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning:
The dedent function was deprecated in Matplotlib 3.1 and will be removed in 3.3.
Use inspect.cleandoc instead.
  This is separate from the ipykernel package so we can avoid doing imports
```

```
until
/home/hzzhyj/.local/lib/python3.7/site-packages/ipykernel_launcher.py:6:
MatplotlibDeprecationWarning:
The dedent function was deprecated in Matplotlib 3.1 and will be removed in 3.3.
Use inspect.cleandoc instead.
```

[378]:
```
# 28.291565 latitude -16.629129 longitude Canary islands
fig = plt.figure(figsize=(6,6))
m = Basemap(llcrnrlon=-18,llcrnrlat=27, urcrnrlon=-13, urcrnrlat=30,␣
 ↪projection='merc', resolution="h", epsg=5520)
#http://server.arcgisonline.com/arcgis/rest/services
m.arcgisimage(service='ESRI_Imagery_World_2D', xpixels = 1500)
x, y = m(longitudes, latitudes)

for i in range(n_clusters):
    plt.scatter(x[aemet_cluster_5==i],
                y[aemet_cluster_5==i],
                label='cluster ' + str(i+1),
                s=80,
                color=cluster_color_cycle[i],
```

```
                edgecolors = 'white')
plt.annotate(aemet['meta'][33][1], (x[33], y[33]), color='red')
plt.show()
```

```
/home/hzzhyj/.local/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning:
The dedent function was deprecated in Matplotlib 3.1 and will be removed in 3.3.
Use inspect.cleandoc instead.
  This is separate from the ipykernel package so we can avoid doing imports
until
```

[ ]:

2 Phonemes dataset

[64]:
```
from skfda.exploratory.depth import *
from skfda.preprocessing.smoothing import BasisSmoother
def depth_based_median(fdatagrid, depth_method=modified_band_depth):
    """Compute the median based on a depth measure.

    The depth based median is basically the deepest curve given a certain
    depth measure

    Args:
        fdatagrid (FDataGrid): Object containing different samples of a
            functional variable.
        depth_method (:ref:`depth measure <depth-measures>`, optional):
                Method used to order the data. Defaults to :func:`modified
                band depth <fda.depth_measures.modified_band_depth>`.

    Returns:
        FDataGrid: object containing the computed depth_based median.
```

```
    """
    depth = depth_method(fdatagrid)
    indices_descending_depth = (-depth).argsort(axis=0)

    # The median is the deepest curve
    return fdatagrid[indices_descending_depth[0]]


def trimmed_means(fdatagrid,
                  trimmed_percentage,
                  depth_method=modified_band_depth):
    """Compute the trimmed means based on a depth measure.

    The trimmed means consists in computing the mean function without a
    percentage of least deep curves. That is, we first remove the least deep
    curves and then we compute the mean as usual.

    Args:
        fdatagrid (FDataGrid): Object containing different samples of a
            functional variable.
        trimmed_percentage (float): indicates the percentage of functions to
            remove. It is not easy to determine as it varies from dataset to
            dataset.
        depth_method (:ref:`depth measure <depth-measures>`, optional):
            Method used to order the data. Defaults to :func:`modified
            band depth <fda.depth_measures.modified_band_depth>`.

    Returns:
        FDataGrid: object containing the computed trimmed mean.

    """
    n_samples_to_keep = int((fdatagrid.n_samples -
                         fdatagrid.n_samples * trimmed_percentage))

    # compute the depth of each curve and store the indexes in descending order
    depth = depth_method(fdatagrid)
    indices_descending_depth = (-depth).argsort(axis=0)

    trimmed_curves = fdatagrid[indices_descending_depth[:n_samples_to_keep]]

    return trimmed_curves.mean()

def get_curves_ordered_by_depth_trimmed(fdatagrid,
                                        percentage,
                                        depth_method=modified_band_depth):
    n_samples_to_keep = int((fdatagrid.n_samples -
```

```
                        fdatagrid.n_samples * percentage))

    # compute the depth of each curve and store the indexes in descending order
    depth = depth_method(fdatagrid)
    indices_descending_depth = (-depth).argsort(axis=0)

    return fdatagrid[indices_descending_depth[:n_samples_to_keep]]
```

[65]:
```
def get_mean_functions(fd, names, extra_data=False):
    means = []
    for i in range(len(names)):
        means.append(fd[y==i].mean())
        means[i].dataset_label = names[i]
        if extra_data:
            means[i] = means[i].concatenate(fd[y==i][:5])
    means.append(fd.mean())
    return means
```

[66]:
```
def plot_grid(data_list, names, axes_labels=None, ylim=(0,25)):
    if len(data_list) < 6:
        print('data list should have more than 6 elements')
        return
    fig, axs = plt.subplots(3, 2, figsize=(14,14))
    fig.subplots_adjust(hspace=0.3, wspace=0.2)
    counter = 0
    for i in range(3):
        for j in range(2):
            data_list[counter].plot(axs[i,j])
            axs[i,j].set_title(names[0])
            axs[i,j].set_ylim(ylim)
            counter += 1
    if axes_labels is not None:
        for i in range(3):
            for j in range(2):
                axs[i,j].set_xlabel(axes_labels[0])
                axs[i,j].set_ylabel(axes_labels[1])
    plt.suptitle(None)
    plt.show()
```

[67]:
```
def plot_same_fig(data_list, names, length, axes_labels = None, title=None,␣
↪bbox_to_anchor=None, plot_legend=True):
    fig = plt.figure(figsize=(10,6))
    for i in range(length):
        data_list[i].plot(fig, label=names[i])
    if plot_legend:
        if bbox_to_anchor is not None:
            fig.legend(bbox_to_anchor=bbox_to_anchor)
```

```
        else:
            fig.legend()
    plt.title(title)
    plt.suptitle(None)
    if axes_labels is not None:
        plt.xlabel(axes_labels[0])
        plt.ylabel(axes_labels[1])
    plt.show()
```

[68]:
```
def get_three_components_basis(fd_basis, reg_param=0, derivative_degree=2):
    fpca_basis = FPCABasis(n_components=3,
                           regularization_parameter = reg_param,
                           penalty=derivative_degree)
    fpca_basis.fit(fd_basis.copy(coefficients=fd_basis.coefficients.copy()))
    return fpca_basis.components_
```

[69]:
```
def get_three_components_grid(fd_grid, reg_param=0, derivative_degree=2):
    fpca_grid = FPCAGrid(n_components=3,
                           regularization_parameter = reg_param,
                           penalty=derivative_degree)
    fpca_grid.fit(fd_grid.copy(data_matrix=np.squeeze(fd_grid.data_matrix).
 ↪copy()))
    return fpca_grid.components_
```

[70]:
```
def get_smoothed_dataset(fd, smoothing_parameter=0):
    smoother = BasisSmoother(basis=BSpline(n_basis=55, domain_range=fd.
 ↪domain_range),
                             return_basis=True,
                             smoothing_parameter=smoothing_parameter,
                             penalty=2)
    fd_basis = smoother.fit_transform(fd)
    fd_basis.axes_labels = fd.axes_labels
    return fd_basis
```

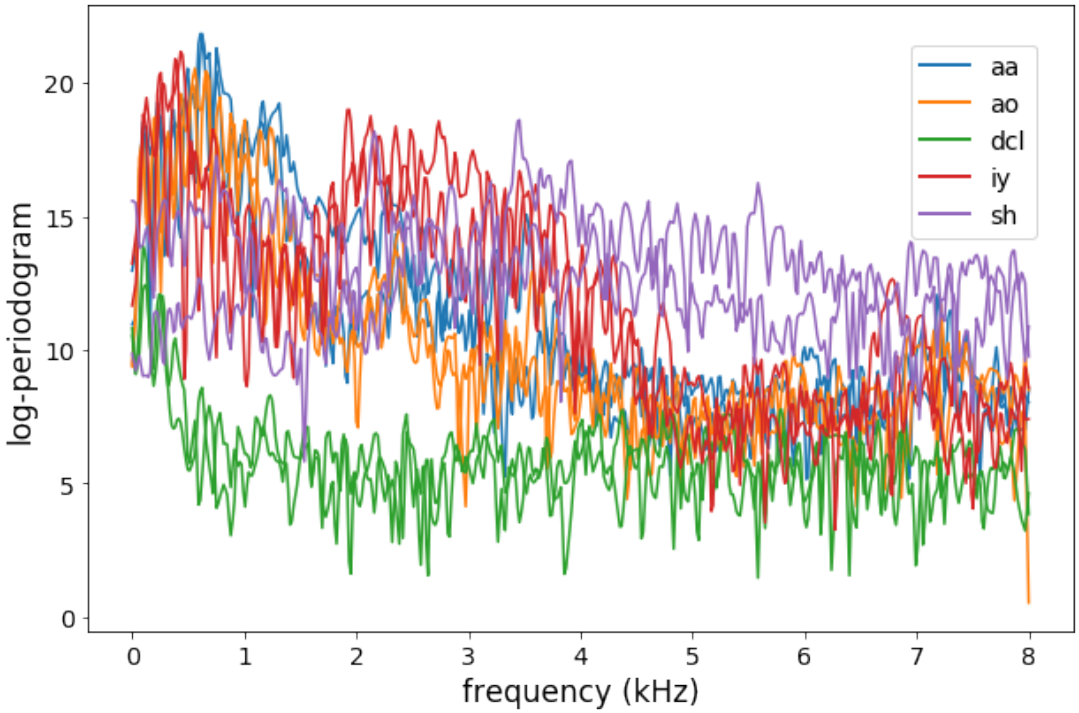
[ ]:
```
```

[71]:
```
phonemes = fetch_phoneme()
y = phonemes['target']
fd = phonemes['data']
names = phonemes['target_names']
```

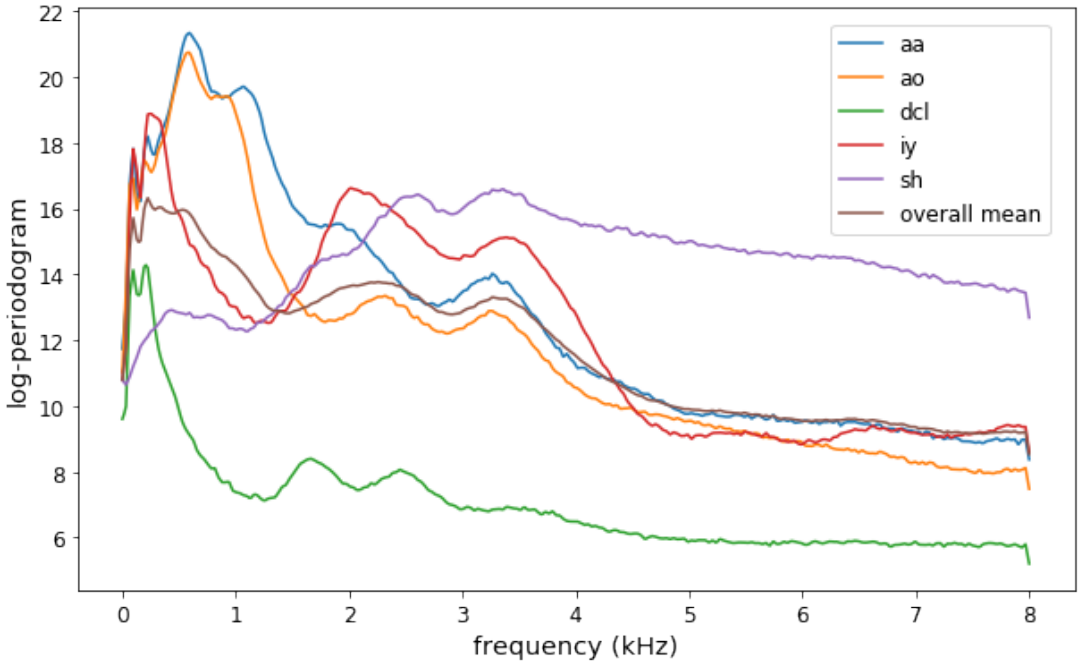
[72]:
```
fd = FDataGrid(data_matrix=np.squeeze(fd.data_matrix),
                   sample_points=np.array(range(0, 256))*8/255,
                   dataset_label="Phoneme",
                   axes_labels=["frequency (kHz)", "log-periodogram"])
```

[73]:
```
aa = fd[y==0]
ao = fd[y==1]
dcl = fd[y==2]
iy = fd[y==3]
sh = fd[y==4]
```

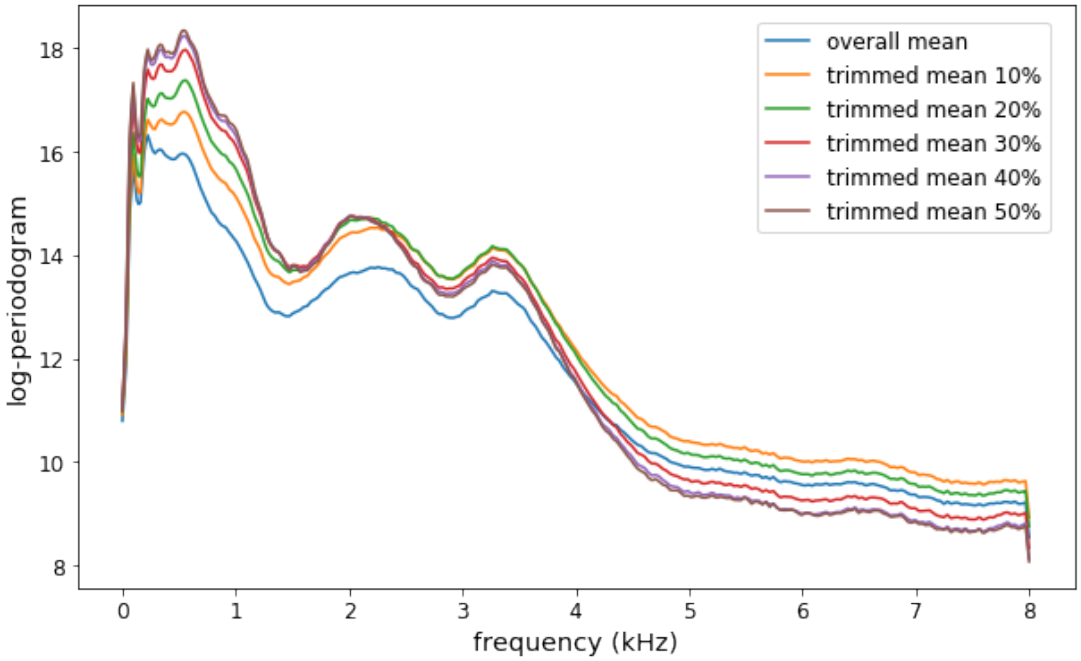
[74]:
```
from matplotlib.lines import Line2D
n_phonemes = 2
fig=plt.figure(figsize=(10,6.5))
aa[:n_phonemes].plot(fig=fig, color='C0', label='aa')
ao[:n_phonemes].plot(fig=fig, color='C1')
dcl[:n_phonemes].plot(fig=fig, color='C2')
iy[:n_phonemes].plot(fig=fig, color='C3')
sh[:n_phonemes].plot(fig=fig, color='C4')

custom_lines = [Line2D([0], [0], color='C0', lw=2),
                Line2D([0], [0], color='C1', lw=2),
                Line2D([0], [0], color='C2', lw=2),
                Line2D([0], [0], color='C3', lw=2),
                Line2D([0], [0], color='C4', lw=2)]
fig.legend(custom_lines, ['aa', 'ao', 'dcl', 'iy', 'sh'],
          bbox_to_anchor=(0.83, 0.83))
plt.suptitle(None)
plt.savefig('phonemes_ten_curves.pdf')
plt.show()
```



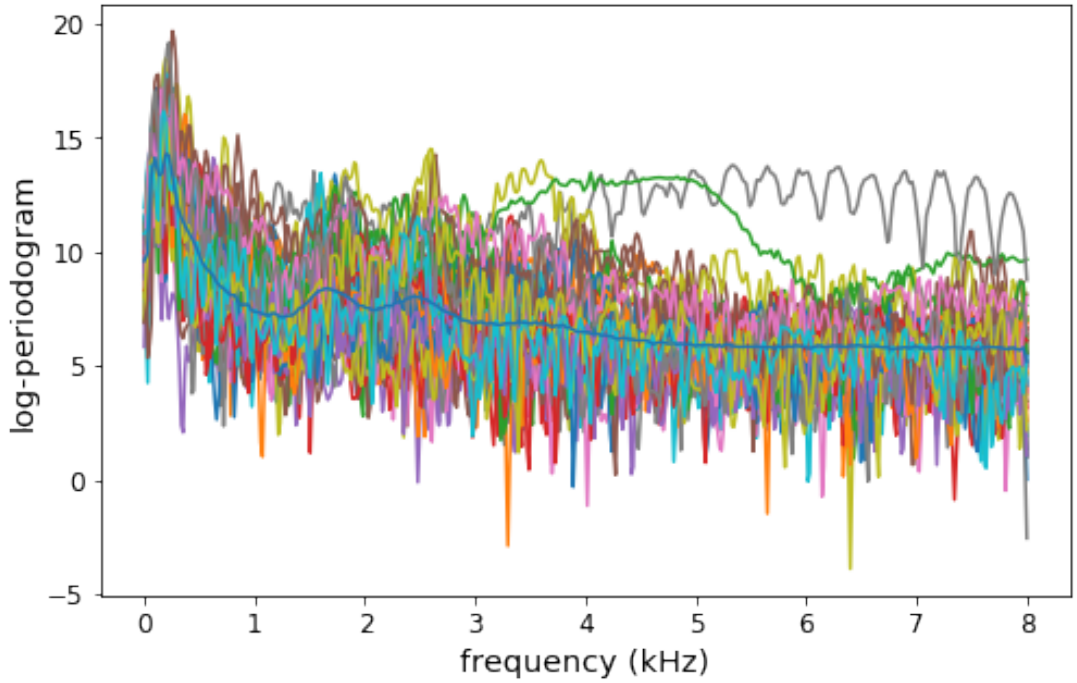
[45]:
```
means = get_mean_functions(fd, names, False)
plot_same_fig(means, names + ['overall mean'], 6, bbox_to_anchor=(0.83, 0.85))
```



[46]:
```
percentages = range(10, 100, 10)
trimmed_means_titles = ['trimmed mean ' + str(i) + '%' for i in percentages]
trimmed_overall_means = [trimmed_means(fd, i/100, fraiman_muniz_depth) for i in␣
 ↪percentages]
n_trimmed_means = 5
plot_same_fig([fd.mean()] + trimmed_overall_means[:n_trimmed_means],
              ['overall mean'] + trimmed_means_titles[:n_trimmed_means],
             n_trimmed_means+1,
             bbox_to_anchor=(0.83,0.85))
```

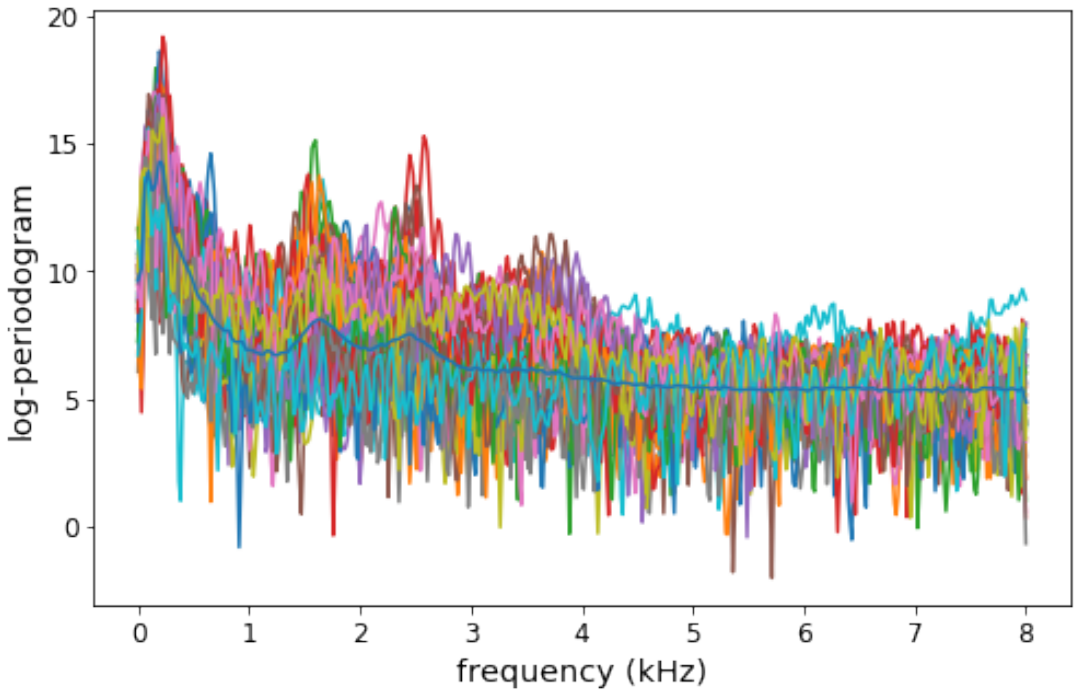


[47]:
```
dcl[np.random.choice(aa.n_samples, 50, replace=False)].concatenate(dcl.mean()).
↪plot()
plt.suptitle(None)
plt.show()
```

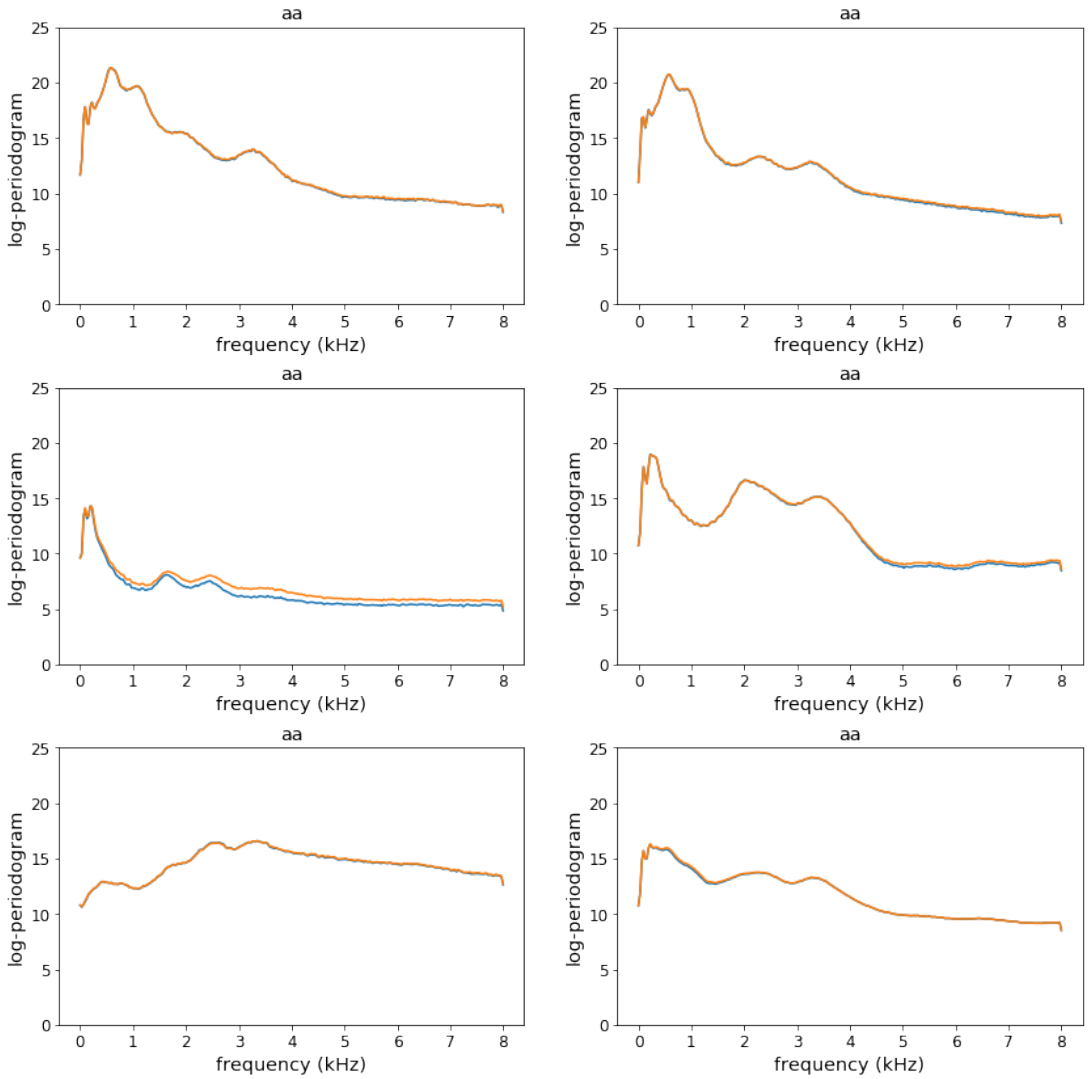


[48]:
```
trim_percentage = 0.2
depth_method = fraiman_muniz_depth
aa_trimmed = get_curves_ordered_by_depth_trimmed(aa, trim_percentage,␣
 ↪depth_method)
aa_trimmed_mean = trimmed_means(aa, trim_percentage, depth_method)
ao_trimmed = get_curves_ordered_by_depth_trimmed(ao, trim_percentage,␣
 ↪depth_method)
ao_trimmed_mean = trimmed_means(ao, trim_percentage, depth_method)
dcl_trimmed = get_curves_ordered_by_depth_trimmed(dcl, trim_percentage,␣
 ↪depth_method)
dcl_trimmed_mean = trimmed_means(dcl, trim_percentage, depth_method)
iy_trimmed = get_curves_ordered_by_depth_trimmed(iy, trim_percentage,␣
 ↪depth_method)
iy_trimmed_mean = trimmed_means(iy, trim_percentage, depth_method)
sh_trimmed = get_curves_ordered_by_depth_trimmed(sh, trim_percentage,␣
 ↪depth_method)
sh_trimmed_mean = trimmed_means(sh, trim_percentage, depth_method)
```

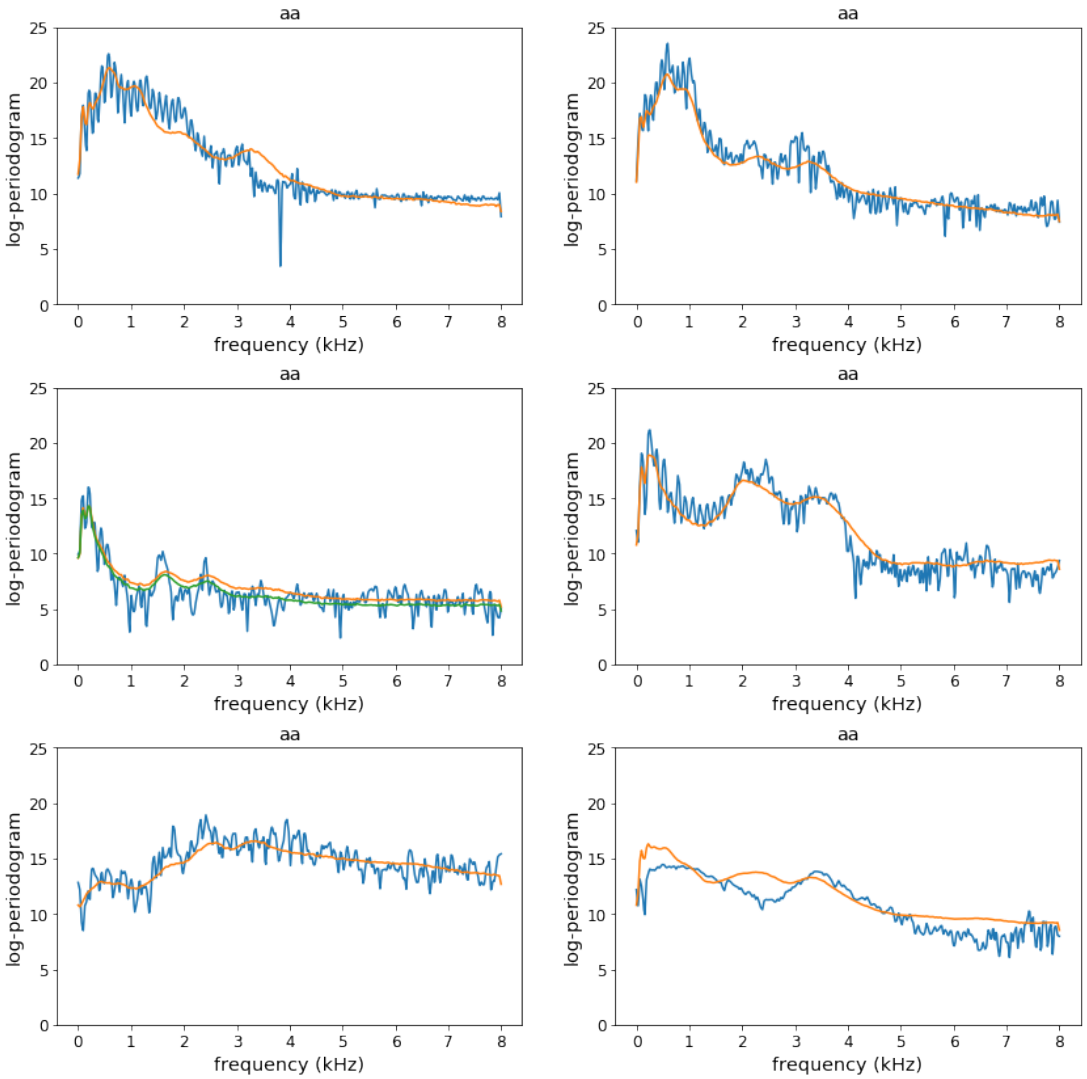
[49]:
```
dcl_trimmed[np.random.choice(dcl_trimmed.n_samples, 50, replace=False)].
 ↪concatenate(dcl_trimmed_mean).plot()
plt.suptitle(None)
plt.show()
```



[50]:
```
trimmed_dataset = aa_trimmed.copy()
trimmed_dataset = trimmed_dataset.concatenate(ao)
trimmed_dataset = trimmed_dataset.concatenate(dcl)
trimmed_dataset = trimmed_dataset.concatenate(iy)
trimmed_dataset = trimmed_dataset.concatenate(sh)
trimmed_dataset_mean = trimmed_dataset.mean()

trimmed_means_list = [
    aa_trimmed_mean,
    ao_trimmed_mean,
    dcl_trimmed_mean,
    iy_trimmed_mean,
    sh_trimmed_mean,
    trimmed_dataset_mean
]
trimmed_means_and_non_trimmed_means = [
    aa_trimmed_mean.copy().concatenate(aa.mean()),
    ao_trimmed_mean.copy().concatenate(ao.mean()),
    dcl_trimmed_mean.copy().concatenate(dcl.mean()),
    iy_trimmed_mean.copy().concatenate(iy.mean()),
    sh_trimmed_mean.copy().concatenate(sh.mean()),
    trimmed_dataset_mean.copy().concatenate(fd.mean())
]
plot_grid(trimmed_means_and_non_trimmed_means, names + ['overall trimmed mean'])
```



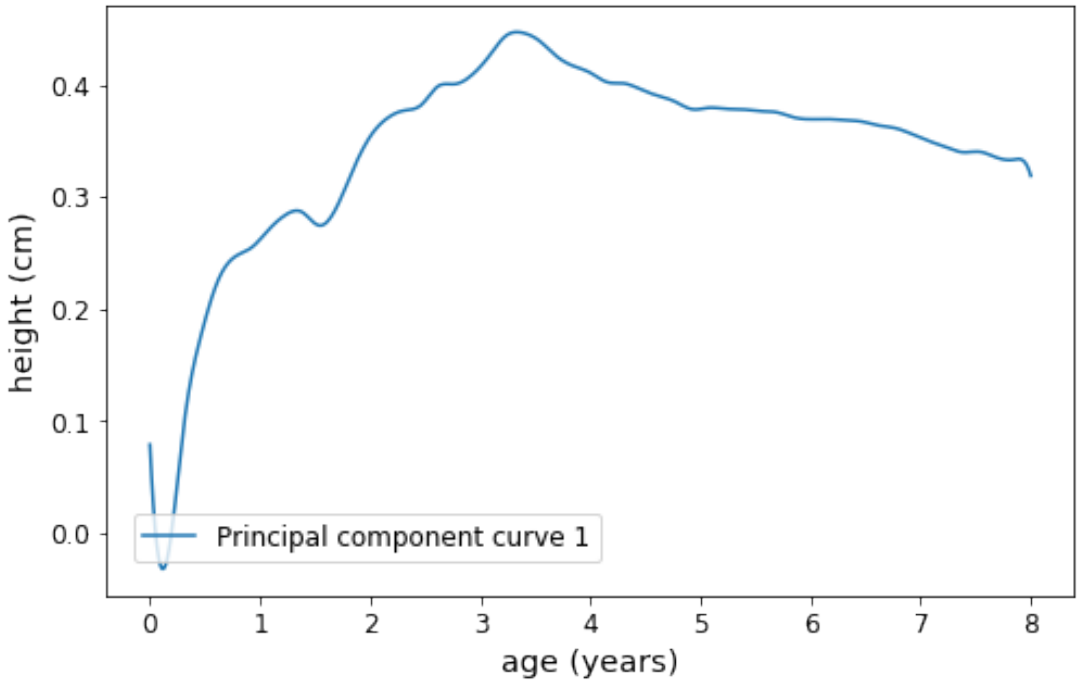
[51]:
```
depth_based_medians = [
    depth_based_median(aa).concatenate(aa.mean()),
    depth_based_median(ao).concatenate(ao.mean()),
    depth_based_median(dcl).concatenate(dcl.mean()).
 ↪concatenate(dcl_trimmed_mean),
    depth_based_median(iy).concatenate(iy.mean()),
    depth_based_median(sh).concatenate(sh.mean()),
    depth_based_median(fd).concatenate(fd.mean())
]
plot_grid(depth_based_medians, names + ['general median'])
```



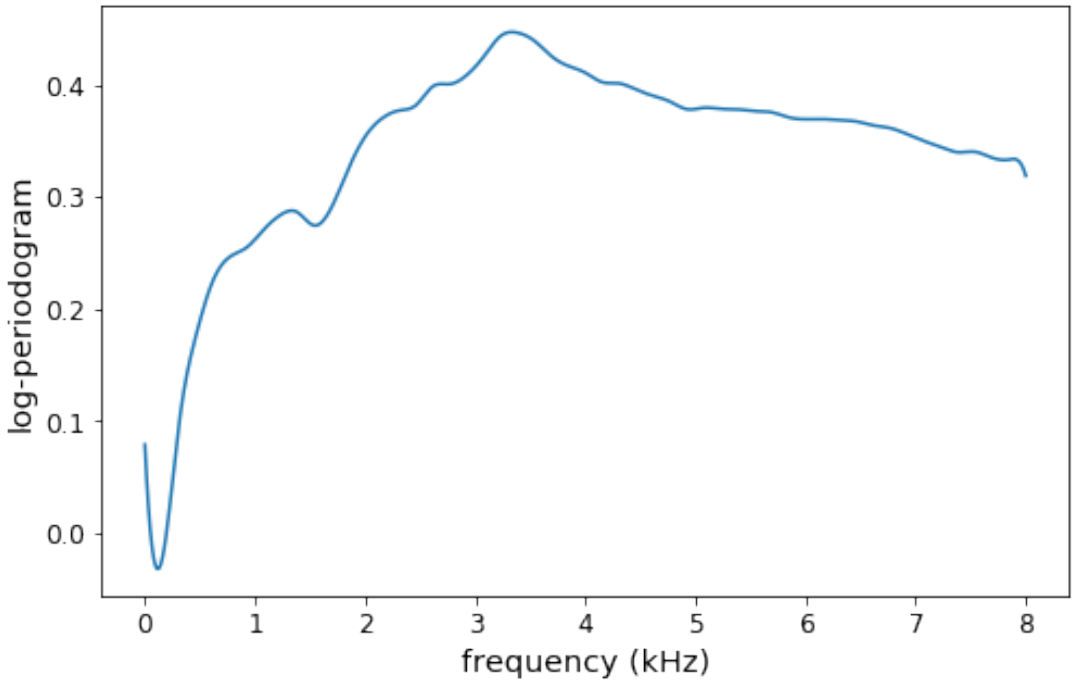
[ ]:

2.1 First principal component

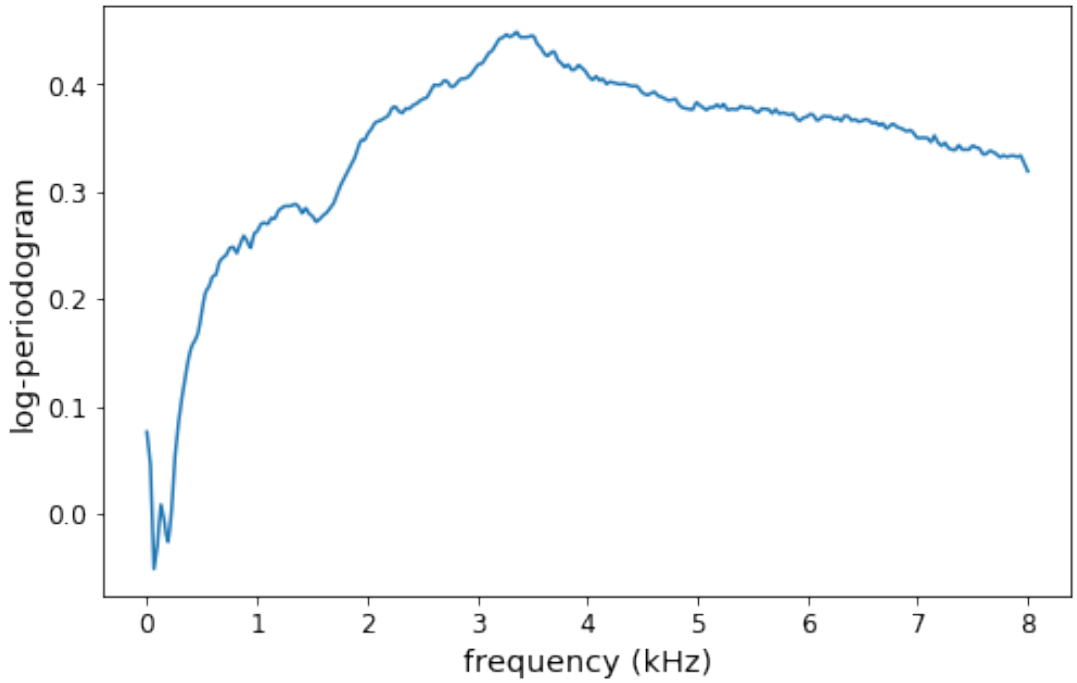
[29]:
```
fpca_basis=FPCA()
fpca_basis.fit(fd.to_basis(BSpline(n_basis=55)))
```



[32]:
```
fpca_basis.components_[0].plot()
plt.xlabel('frequency (kHz)')
plt.ylabel('log-periodogram')
plt.show()
```



[36]:
```
fpca_grid=FPCA()
fpca_grid = fpca_grid.fit(fd)
fpca_grid.components_[0].plot()
plt.suptitle(None)
plt.xlabel('frequency (kHz)')
plt.ylabel('log-periodogram')
plt.show()
```

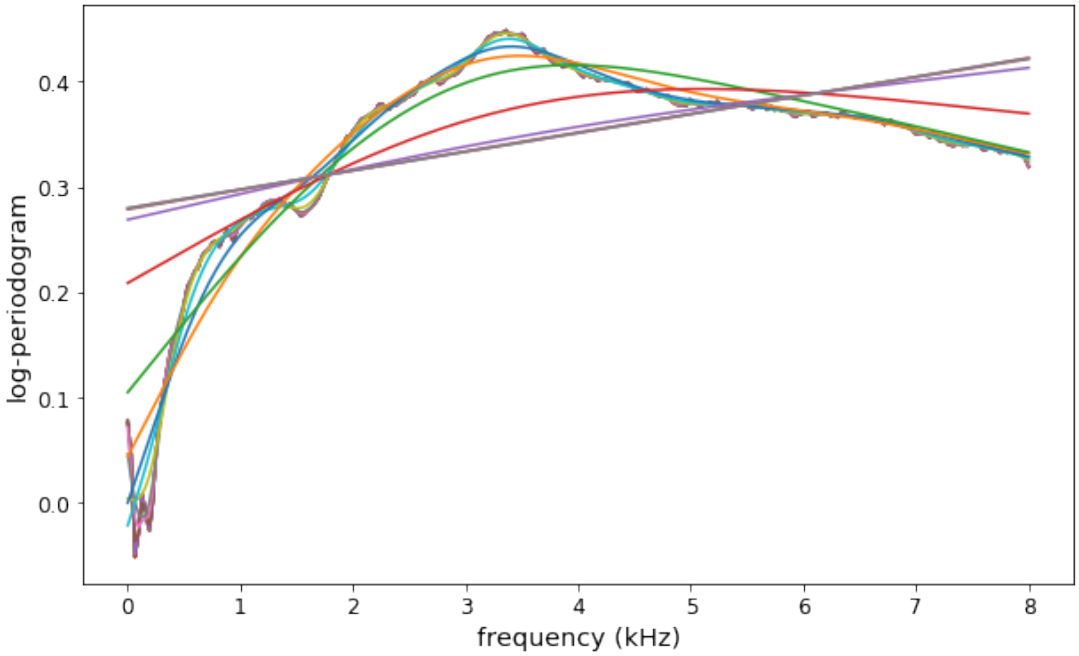


[ ]:

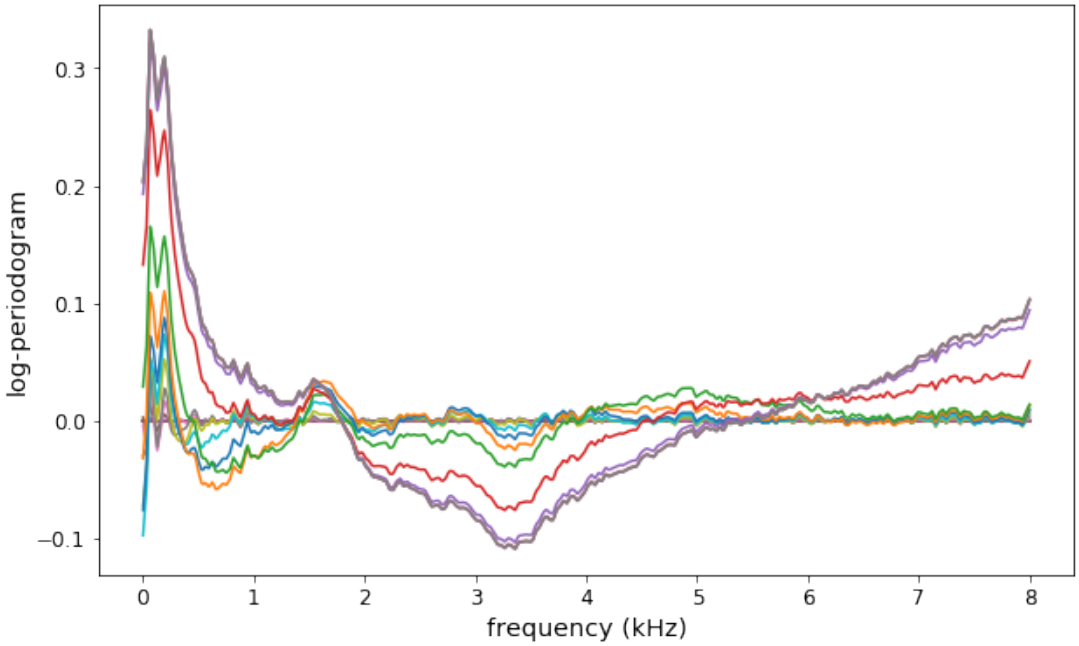
2.2 Regularization

[52]:
```
param_range = range(-5, 12)
params = [(10)**i for i in param_range]
params_names = ['lambda=10^' + str(i) for i in param_range]
components_regularization_grid = []
for i in range(len(params)):
    components_regularization_grid.append(get_three_components_grid(fd,␣
 ↪reg_param=params[i]))
first_components_grid=[component[0] for component in␣
 ↪components_regularization_grid]
no_reg_grid = get_three_components_grid(fd, reg_param=0)[0]
first_components_grid.insert(0, no_reg_grid)
params_names.insert(0, 'component without regularization')
```

[53]:
```
plot_same_fig(first_components_grid, params_names,
              len(first_components_grid),  fd.axes_labels,
              title=None, plot_legend=False)
```

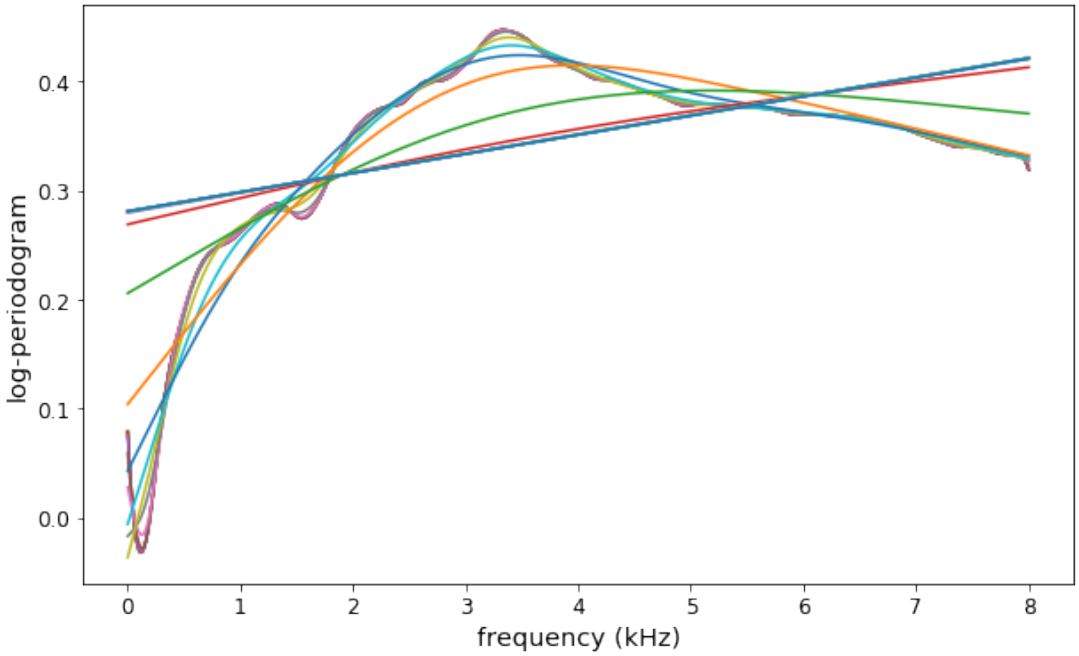


[54]:
```
differences_with_no_reg_grid = [
    first_components_grid[i] - no_reg_grid for i in␣
 ↪range(len(first_components_grid))
]
plot_same_fig(differences_with_no_reg_grid, params_names,␣
 ↪len(first_components_grid),
              fd.axes_labels, plot_legend=False)
```

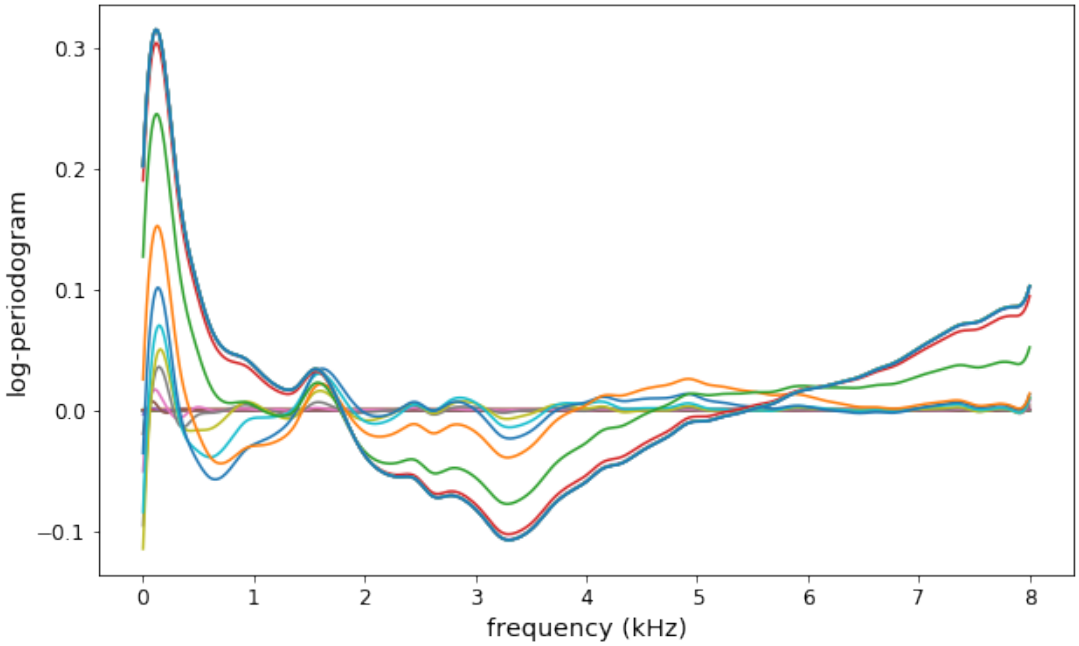


[55]:
```
param_range = range(-10, 10)
params = [(10)**i for i in param_range]
params_names = ['lambda=10^' + str(i) for i in param_range]
fd_basis=fd.to_basis(BSpline(n_basis=55))
components_regularization_basis = []
for i in range(len(params)):
    components_regularization_basis.append(get_three_components_basis(fd_basis,␣
 ↪reg_param=params[i]))
first_components_basis=[component[0] for component in␣
 ↪components_regularization_basis]
no_reg_basis = get_three_components_basis(fd_basis, reg_param=0)[0]
first_components_basis.insert(0, no_reg_basis)
params_names.insert(0, 'component without regularization')
```

[56]:
```
plot_same_fig(first_components_basis, params_names, len(first_components_basis),
              fd.axes_labels, plot_legend=False)
```

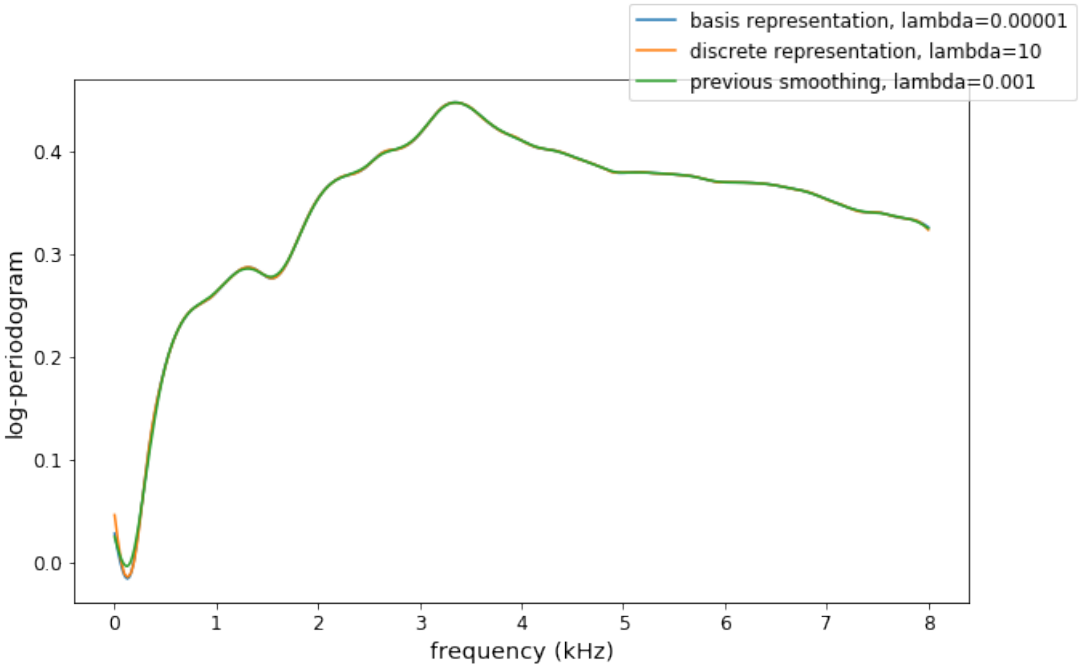


[57]:
```
differences_with_no_reg_basis = [
    first_components_basis[i] - no_reg_basis for i in␣
 ↪range(len(first_components_basis))
]
plot_same_fig(differences_with_no_reg_basis, params_names,␣
 ↪len(first_components_basis),  fd.axes_labels,
            plot_legend=False)
```



[61]:
```
smoothed_datasets = []
for i in range(len(params)):
    smoothed_datasets.append(get_smoothed_dataset(fd,␣
 ↪smoothing_parameter=params[i]))
components_smoothed_datasets = [
    get_three_components_basis(dataset) for dataset in smoothed_datasets
]
first_components_smoothed_datasets = [
    component[0] for component in components_smoothed_datasets
]
```

[65]:
```
component_basis_reg = first_components_basis[6]
component_grid_reg = first_components_grid[7]
component_basis_no_reg = first_components_smoothed_datasets[7]
final_list =[
    component_basis_reg,
    component_grid_reg,
    component_basis_no_reg
]
final_list_names = [
    'basis representation, lambda=0.00001',
    'discrete representation, lambda=10',
    'previous smoothing, lambda=0.001'
]
plot_same_fig(final_list, final_list_names, len(final_list),  fd.axes_labels)
```


basis representation, lambda=0.00001
discrete representation, lambda=10
previous smoothing, lambda=0.001
log-periodogram
0.0
0.1
0.2
0.3
0.4
0
1
2
3
4
5
6
7
8
frequency (kHz)

[2]:

C

FPCA Documentation

In this appendix the documentation related with the FPCA module is shown. This includes documentation related to the Preprocessing module, the submodule Dimensionality Reduction and the class FPCA.

Preprocessing

Sometimes we need to preprocess the data prior to analyze it. The modules in this category deal with this problem.

Smoothing

If the functional data observations are noisy, *smoothing* the data allows a better representation of the true underlying functions. You can learn more about the smoothing methods provided by scikit-fda here.

Registration

Sometimes, the functional data may be misaligned, or the phase variation should be ignored in the analysis. To align the data and eliminate the phase variation, we need to use *registration* methods. Here you can learn more about the registration methods available in the library.

Dimensionality Reduction

The functional data may have too many features so we cannot analyse the data with clarity. To better understand the data, we need to use *dimensionality reduction* methods that can reduce the number of features while still preserving the most relevant information. Here you can learn more about the dimension reduction methods available in the library.

Dimensionality Reduction

When dealing with data samples with high dimensionality, we often need to reduce the dimensions so we can better observe the data.

Projection

One way to reduce the dimension is through projection. For example, in functional principal component analysis, we project the data samples into a smaller sample of functions that preserve the maximum sample variance.

Modules:

- Functional Principal Component Analysis (FPCA)
  - FPCA for functional data in both representations
    - FPCA
      - Examples using `skfda.preprocessing.dim_reduction.projection.FPCA`

Functional Principal Component Analysis (FPCA)

This module provides tools to analyse functional data using FPCA. FPCA is a common tool used to reduce dimensionality. It can be applied to a functional data object in either a basis representation or a discretized representation. The output of FPCA are the projections of the original sample functions into the directions (principal components) in which most of the variance is conserved. In multivariate PCA those directions are vectors. However, in FPCA we seek functions that maximizes the sample variance operator, and then project our data samples into those principal components. The number of principal components are at most the number of original features.

For a detailed example please view Functional Principal Component Analysis, where the process is applied to several datasets in both discretized and basis forms.

FPCA for functional data in both representations

`skfda.preprocessing.dim_reduction.projection.FPCA`([...])	Class that implements functional princ

FPCA

class **skfda.preprocessing.dim_reduction.projection.FPCA**(*n_components=3, centering=True, regularization=None, weights=None, components_basis=None*) [source]

Class that implements functional principal component analysis for both basis and grid representations of the data. Most parameters are shared when fitting a FDataBasis or FDataGrid, except weights and components_basis.

Parameters

- **n_components** (*int*) – number of principal components to obtain from functional principal component analysis. Defaults to 3.
- **centering** (*bool*) – if True then calculate the mean of the functional data object and center the data first. Defaults to True. If True the passed FDataBasis object is modified.
- **regularization** (*Regularization*) – Regularization object to be applied.
- **components_basis** (*Basis*) – the basis in which we want the principal components. We can use a different basis than the basis contained in the passed FDataBasis object. This parameter is only used when fitting a FDataBasis.
- **weights** (*numpy.array or callable*) – the weights vector used for discrete integration. If none then the trapezoidal rule is used for computing the weights. If a callable object is passed, then the weight vector will be obtained by evaluating the object at the sample points of the passed FDataGrid object in the fit method. This parameter is only used when fitting a FDataGrid.

components_

this contains the principal components in a basis representation.

Type

FDataBasis

explained_variance_

The amount of variance explained by each of the selected components.

Type

array_like

explained_variance_ratio_

this contains the percentage of variance explained by each principal component.

Type

array_like

Examples

Construct an artificial FDataBasis object and run FPCA with this object. The resulting principal components are not compared because there are several equivalent possibilities.

```
>>> data_matrix = np.array([[1.0, 0.0], [0.0, 2.0]])
>>> sample_points = [0, 1]
>>> fd = FDataGrid(data_matrix, sample_points)
>>> basis = skfda.representation.basis.Monomial((0,1), n_basis=2)
>>> basis_fd = fd.to_basis(basis)
>>> fpca_basis = FPCA(2)
>>> fpca_basis = fpca_basis.fit(basis_fd)
```

In this example we apply discretized functional PCA with some simple data to illustrate the usage of this class. We initialize the FPCA object, fit the artificial data and obtain the scores. The results are not tested because there are several equivalent possibilities.

```
>>> data_matrix = np.array([[1.0, 0.0], [0.0, 2.0]])
>>> sample_points = [0, 1]
>>> fd = FDataGrid(data_matrix, sample_points)
>>> fpca_grid = FPCA(2)
>>> fpca_grid = fpca_grid.fit(fd)
```

Methods

__init__([n_components, centering, ...])	Initialize self.
fit(X[, y])	Computes the n_components first principa
FPCA.fit_basis	
FPCA.fit_grid	
fit_transform(X[, y])	Computes the n_components first principa
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Computes the n_components first principa

`FPCA.transform_basis`	
`FPCA.transform_grid`	

`__init__`(*n_components=3*, *centering=True*, *regularization=None*, *weights=None*, *components_basis=None*) [source]

Initialize self. See help(type(self)) for accurate signature.

`fit`(*X*, *y=None*) [source]

Computes the n_components first principal components and saves them inside the FPCA object, both FDataGrid and FDataBasis are accepted

Parameters

- **X** (*FDataGrid or FDataBasis*) – the functional data object to be analysed
- **y** (*None*, *not used*) – only present for convention of a fit function

Returns

self (object)

`fit_transform`(*X*, *y=None*, ***fit_params*) [source]

Computes the n_components first principal components and their scores and returns them. :param X: the functional data object to be analysed :type X: FDataGrid or FDataBasis :param y: only present for convention of a fit function :type y: None, not used

Returns

the scores of the data with reference to the principal components

Return type

(array_like)

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep (*bool*, *default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params – Parameter names mapped to their values.

Return type

mapping of string to any

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** (*dict*) – Estimator parameters.

Returns

self – Estimator instance.

Return type

object

transform(*X*, *y=None*) [source]

Computes the n_components first principal components score and returns them.

Parameters

- **X** (*FDataGrid or FDataBasis*) – the functional data object to be analysed
- **y** (*None*, *not used*) – only present because of fit function convention

Returns

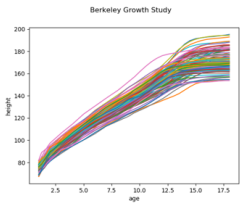
the scores of the data with reference to the principal components

Return type

(array_like)

Examples using

`skfda.preprocessing.dim_reduction.projection.FPCA`



Functional Principal